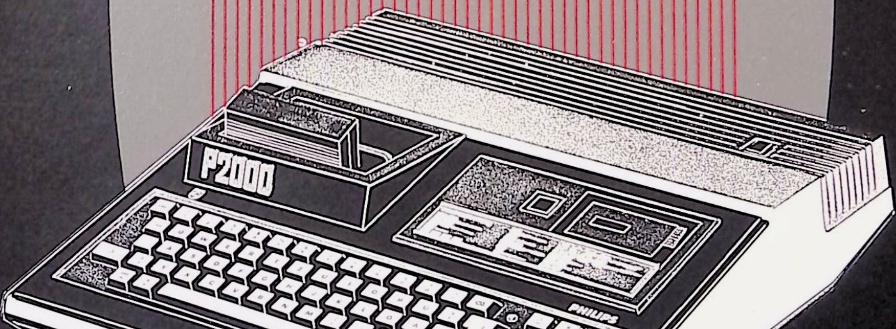
PHILIPS

P2000

P2306
DISK BASIC
REFERENCE MANUAL


P2000
PHILIPS

**Data
Systems**

PHILIPS

P2306

DISK BASIC
REFERENCE MANUAL

IMPORTANT NOTE:

This software program is distributed on an
'as is' basis without any warranty or liability.

COPYRIGHT BY PHILIPS, 1981, all rights reserved.

5103 991 10625

PREFACE

This manual is a reference manual for the P2000 DISK BASIC programming language. It is intended for reference purposes only and should be used by experienced BASIC programmers. It can also be used by the person who has recently completed some form of study of BASIC, but in no way should it be seen or used as a learning book for the language.

Chapter 1 contains an introduction to BASIC, covering the General Working Environment of a BASIC Interpreter and start-up procedures for use on the P2000.

Chapter 2 deals with general information about BASIC. It covers such things as modes, line format, types of variables and constants etc.

Chapter 3 contains details of the commands and functions in alphabetical sequence. You will find a description of the notations used in the early parts of the chapter.

Chapter 4 provides information about file handling, giving all the file-types available, their associated commands and how to create and use them.

Chapter 5 has been included especially for the user who is interested in the way that BASIC uses the memory. It includes an overall view of the memory; it looks at how numerics and strings are represented; and it shows how you can interface between BASIC and your own Z80 machine code sub-routines.

Appendixes A-I at the back of the manual contain various pieces of useful information. It is well worthwhile making yourself aware of their contents before continuing with your applications.

PAGE LAYOUT

The pages in all P2000 manuals are arranged as follows:

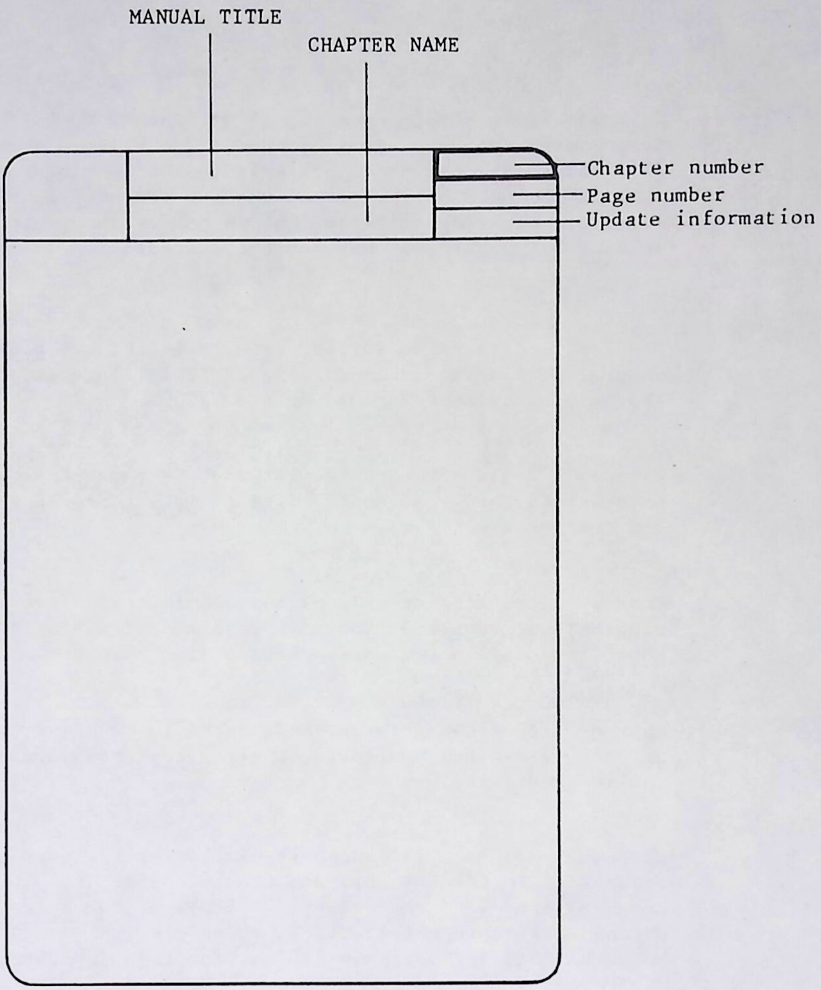

MANUAL TITLE
CHAPTER NAME
Chapter number
Page number
Update information

TABLE OF CONTENTS

page

PREFACE iii

PAGE LAYOUT iv

TABLE OF CONTENTS v

1 INTRODUCTION 1-1
1.1 General Working Environment 1-2
General Working Environment 1-3
1.2 Starting-up the BASIC System 1-4

2 GENERAL INFORMATION ABOUT BASIC 2-1
2.1 Direct/Indirect Modes 2-1
2.2 Line Format 2-2
2.3 Character Set 2-3
Character Set 2-4
2.4 Constants 2-5
2.5 Variables 2-7
2.5.1 Variable Names and Declaration Characters 2-7
2.5.2 Array Variables 2-8
2.6 Type Conversion 2-9
2.7 Expressions and Operators 2-10
2.7.1 Arithmetic Operators 2-10
2.7.2 Relational Operators 2-12
2.7.3 Logical Operators 2-13
Logical Operators 2-14
2.7.4 Functional Operators 2-15
2.7.5 String Operations 2-15

page

2.8 Input Editing 2-16

3 BASIC COMMANDS AND FUNCTIONS 3-1
Format Notation 3-2
3.2 ABS 3-3
3.3 ASC 3-3
3.4 ATN 3-4
3.5 AUTO 3-5
3.6 CALL 3-6
3.7 CDBL 3-7
3.8 CHAIN 3-8
CHAIN 3-9
3.9 CHR$ 3-10
3.10 CINT 3-10
3.11 CLEAR 3-11
3.12 CLOSE 3-12
3.13 COMMON 3-13
3.14 CONT 3-14
3.15 COS 3-15
3.16 CSNG 3-16
3.17 CVI, CVS, CVD 3.17
3.18 DATA 3-18
3.19 DEF FN 3-19
DEF FN 3-20
3.20 DEFINT/SNG/DEL/STR 3-21
3.21 DEF USR 3-22
3.22 DELETE 3-23
3.23 DIM 3-24
3.24 EDIT 3-25
EDIT 3-26
EDIT 3-27
EDIT 3-28
3.25 END 3-29
3.26 EOF 3-30
3.27 ERASE 3-31
3.28 ERR AND ERL VARIABLES 3-32
3.29 ERROR 3-33
ERROR 3-34
3.30 EXP 3-35
3.31 FIELD 3-36
3.32 FIX 3-37
3.33 FOR...NEXT 3-38
FOR...NEXT 3-39
3.34 FRE 3-40
3.35 GET 3-41
3.36 GOSUB 3-42
3.37 GOTO 3-43
3.38 HEX$ 3-44

page

3.39 IF...THEN...ELSE 3-45
IF...THEN...ELSE 3-46
3.40 INKEY$ 3-47
3.41 INP 3-48
3.42 INPUT 3-49
INPUT 3-50
3.43 INPUT# 3-51
3.44 INPUT$ 3-52
3.45 INSTR 3-53
3.46 INT 3-53
3.47 KILL 3-54
3.48 LEFT$ 3-55
3.49 LEN 3-55
3.50 LET 3-56
3.51 LINE INPUT 3-57
3.52 LINE INPUT# 3-58
3.53 LIST 3-59
LIST 3-60
3.54 LLIST 3-61
3.55 LOAD 3-62
3.56 LOC 3-63
3.57 LOG 3-64
3.58 LPOS 3-65
3.59 LPRINT AND LPRINT USING 3-66
3.60 LSET AND RSET 3-67
3.61 MERGE 3-68
3.62 MID$ 3-69
3.63 MID$ 3-70
3.64 MKI$, MKS$, MKD$ 3-71
3.65 NAME 3-72
3.66 NEW 3-73
3.67 OCT$ 3-74
3.68 ON ERROR GOTO 3-75
3.69 ON...GOSUB AND ON ...GOTO 3-76
3.70 OPEN 3-77
3.71 OPTION BASE 3-78
3.72 OUT 3-79
3.73 PEEK 3-79
3.74 POKE 3-80
3.75 POS 3-81
3.76 PRINT 3-82
PRINT 3-83
3.77 PRINT USING 3-84
PRINT USING 3-85
PRINT USING 3-86
PRINT USING 3-87
3.78 PRINT# AND PRINT# USING 3-88
PRINT# AND PRINT# USING 3-89
3.79 PUT 3-90
3.80 RANDOMIZE 3-91
3.81 READ 3-92
READ 3-93
3.82 REM 3-94
3.83 RENUM 3-95
3.84 RESET 3-96
3.85 RESTORE 3-97
3.86 RESUME 3-98

page

3.87 RETURN 3-99
3.88 RIGHT$ 3-100
3.89 RND 3-100
3.90 RUN 3-101
3.91 RUN 3-102
3.92 SAVE 3-103
3.93 SGN 3-104
3.94 SIN 3-104
3.95 SPACE$ 3-105
3.96 SPC 3-105
3.97 SQR 3-106
3.98 STOP 3-107
3.99 STRING$ 3-108
3.100 STR$ 3-108
3.101 SWAP 3-109
3.102 TAB 3-110
3.103 TAN 3-110
3.104 TRON/TROFF 3-111
3.105 USR 3-112
3.106 VAL 3-112
3.107 VARPTR 3-113
3.108 WAIT 3-114
3.109 WHILE...WEND 3-115
3.110 WIDTH 3-116
3.111 WRITE 3-117
3.112 WRITE# 3-118

4 DATA FILE HANDLING 4-1
4.1 File Commands 4-1
4.2 Sequential Files 4-5
4.3 Random Files 4-9

5 BASIC'S USE OF MEMORY

5.1 Memory Layout 5-2
5.2 Source Text 5-3
5.3 Variables and Arrays 5-3
  5.3.1 Variables 5-4
    Variables 5-5
    Variables 5-6
  5.3.2 Arrays 5-7
  5.3.3 String Area 5-7
  5.3.4 Sample Program 5-8
  5.3.5 Sample Output 5-9
5.4 The Use of Machine Code Routines 5-10
  5.4.1 Setting a Machine Code Routine 5-11
  5.4.2 The USR Function 5-12
    The USR Function 5-13

APPENDIX A Reserved Words

APPENDIX B Screen and Cursor Handling

APPENDIX C Error Messages

APPENDIX D Mathematical Functions

APPENDIX E ASCII Character Codes

APPENDIX F Useful Adresses

APPENDIX G Keyword Summary

APPENDIX H Programming Efficiency

APPENDIX I The VOLORG Utility

Index

1 INTRODUCTION

This chapter provides an introduction to BASIC on the Philips P2000 micro-computer. It is in fact a Microsoft BASIC and so, if you have worked with one before you will be familiar with the contents of this manual. Section 1.2 contains information that is particularly for use on the P2000, whereas section 1.1 is an overview of the general working environment of the BASIC Interpreter.

The BASIC Interpreter has been supplied in two parts, both of which are necessary to run successfully. Part of the Interpreter is resident within the ROM-key and the remainder of the system software comes on the system disk. Starting up the system is explained in section 1.2, but certainly one of the first things you should do is to make a work copy of your disk. Your master disk should then be stored away in a safe place.

The second disk which is supplied with this package contains the Volume Organization Utility. The details of this utility can be found in Chapter 4 but for now all you need to use is function 3 to copy a complete disk. You should also remember to take regular back-up copies of your program disks; a discipline which is very advisable to adopt.

1.1 GENERAL WORKING ENVIRONMENT

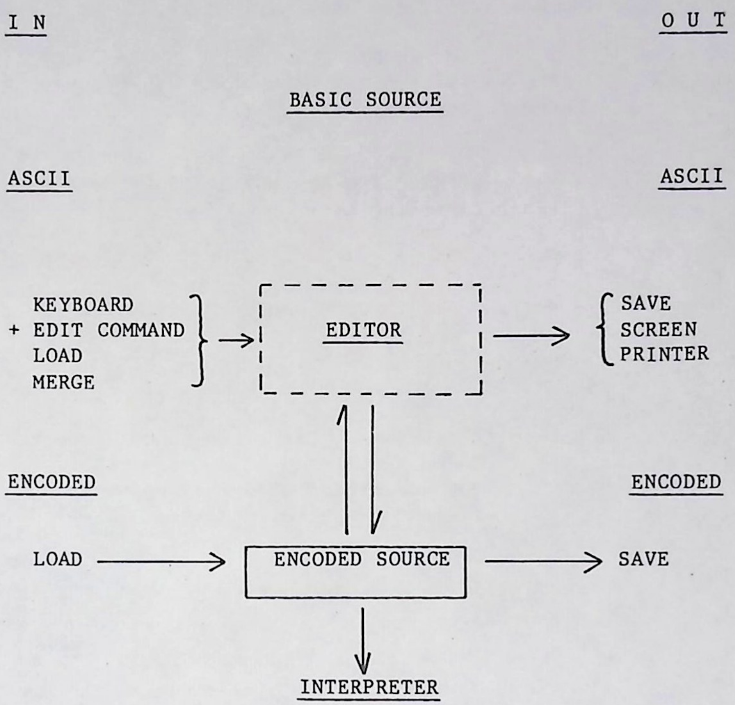
Depending on the environment and language, a program may have to be 'compiled' before it can be executed on the computer. The compiler works on a source module (the written program) and produces an object module in the language of the computer (machine code). Of course there may be other activities which take place before it reaches its final state, however, the significance is that the source and executable versions are completely different. Whenever changes are made, the whole process must be repeated.

BASIC itself can be compiled, but the P2000 BASIC is executed via an Interpreter. In this case the source is also the executable version. Each BASIC statement is interpreted and executed, then process carries on with the next logical statement. If a Basic command is encountered 100 times during the execution of a program, it is interpreted and executed 100 times. It is this concept which allows program execution to be started from any line; it allows the program to be suspended, values checked directly (or even changed to force a result) and execution resumed. Changes to the source can be made directly and so the testing cycle becomes much easier and faster.

Having completed this section, you may find Chapter 5 of this manual very interesting. It covers the memory layout and, among other things, the way that source code is held in memory. In fact, it can help you gain a better understanding of the concepts discussed in this section. But for now let us continue with the general working environment.

The source text that you see on the screen is just a pictorial representation of the way it is held in memory. It is actually held in encoded form in the memory. This encoded form is the only 'true' format of the source and is the part which is interpreted (again, see Chapter 5 for source layout). Any representation of the source, whether it be a display on the screen, a hard copy list or a saved program on disk, is always produced via the Interpreter's Editor function. This is transparent to the user but the knowledge is vital in order to get a detailed understanding of the ways of the Interpreter.

Now let us look at the BASIC Interpreter side in a little more detail:


IN
OUT
BASIC SOURCE
ASCII
ASCII
KEYBOARD
+ EDIT COMMAND
LOAD
MERGE
EDITOR
SAVE
SCREEN
PRINTER
ENCODED
ENCODED
LOAD
ENCODED SOURCE
SAVE
INTERPRETER

Note - The MERGE can only be used on a program which has been saved in ASCII format:

```
SAVE "........",A
```

1.2 START-UP

Firstly, you should refer to the P2000 Information Manual, Number 30121, for instructions regarding the installation of your P2000.

BASIC is initialized automatically at power-up. Follow these instructions:

- Put the BASIC ROM-Key Number P2306 or Number P2311 into the front cartridge slot
- Put the BASIC System Diskette into Disk Drive 1
- Switch on the P2000 Monitor/Television Set
- Switch on the P2000

Note - If the ROM-Key is extracted or inserted while the P2000 is switched on, unpredictable results may occur; i.e. undefined output may be written to the printer or, if a diskette is loaded, it may be erased.

The following message appears on the screen:

```
 PHILIPS DISK BASIC

 Release 1.n UK

 18815 Bytes free


Ok
```

Note - The number of Bytes free depends on the configuration.
n = latest update of the release.

2 GENERAL INFORMATION ABOUT BASIC

2.1 DIRECT/INDIRECT MODES

When BASIC is initialized, it types the prompt "OK". "OK" means BASIC is at command level, that is, it is ready to accept commands. At this point, BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC program lines are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

2.2 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

```
[nnnnn] BASIC statement [: BASIC statement ...] <carriage
return>
```

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon (:).

A BASIC program line always begins with a line number, end with a carriage return and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the terminal's auto-linefeed. Auto-linefeed lets you continue typing a logical line on the next physical line without typing a carriage-return.

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line Numbers must be in the range 0 to 65529. A Period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

2.3 CHARACTER SET

The BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

These special characters are recognized by BASIC:

Character:	Name:
<	Less than
>	Greater than
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up arrow or exponentiation symbol
'	Apostrophe or Integer Division symbol

These special keys are recognized by BASIC:

Keys:	Name:
CODE	Escapes Edit Mode subcommands.
TAB	Moves print position to next tab stop. Tab stops are every eight columns.
⌧	Deletes the line that is currently being typed.
⇧ TAB	Transfers keyboard input to typewriter mode.
⇧ STOP .	Suspends program execution; BASIC will return to input command mode. A suspended program can be resumed by entering the RUN command.
⇧ % 5	Suspends program (5 on numeric pad).
⇧ START 3	Resumes a suspended program.
⇧) 9	Enters EDIT mode on line currently being typed in. (9 on numeric pad).
⇧ = 0	Suppress or resume program output (0 on numeric pad).

2.4 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

"HELLO"
"$ 25,000.00"
"Number of Employees"

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants. Whole numbers between -32768 and + 32767. Integer constants do not have decimal points.

   Example: -6342, +758

2. Fixed Point Constants. Positive or negative real numbers; i.e. numbers that contain decimal points.

   Example: +23.94, -1.643

3. Floating Point Constants. Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally-signed integer (the exponent). The allowable range for floating point constants is 10^{-38} to 10^{+37}.

   Examples:

   235.988E-7= .0000235988
   2359E6 = 2359000000

   (Double precision floating point constants use the letter D instead of E).

4. Hex constants. Hexadecimal numbers with the prefix &H. The allowable range is &H0000 to &HFFFF.

   Examples:

   &H76
   &H32F

5. Octal constants. Octal numbers with the prefix & or &O (ampersand followed by the letter 'O'). The allowable range is &O000000 to &O177777.

   Examples:

   &O347
   &1234

2.4.1 Single and Double Precision Form for Numeric Constants

Numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits. With single precision, the numbers are stored with 7 digits of precision and printed with up to 6 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or

2. exponential form using E, or

3. a trailing Exclamation Mark (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or

2. exponential form using D, or

3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
- 7.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

2.5 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, integer, (real) single precision and double precision variables are assumed to be zero; string variables are assumed to be a zero-length string (i.e."").

2.5.1 Variable Names and Declaration Characters

The first 40 characters of a BASIC variable name, excluding any declaration characters, are significant. The characters allowed in a variable name are letters and digits. The first character must be a letter. Special type declaration characters are also allowed and are a significant part of the variable name - see below. If a special type declaration character is used, the first two characters of the variable name together with the special type declaration character are significant, except the default type declaration character is used.

If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC commands, statements, function names and operator names. Reserved words may not be imbedded.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

$	String variable
%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a variable name is numeric, single precision.

EXAMPLES OF BASIC VARIABLE NAMES:

PI#	declares a double precision value
MI!	declares a single precision value
LI%	declares an integer value
NN$	declares a string value
AB	represents a single precision value

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 3.

2.5.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example for V (10) the subscript range is from 0 to 10. V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is theoretically 255 and practically limited by line length (124). The maximum number of elements per dimension is 32767 in theory and, in practice, is dictated by the amount of memory available. If an array is subscripted by a single-precision or double-precision expression, the subscript is converted to integer (implicit conversion). The expression is rounded, not truncated.

2.6 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs). For example:

   ```
   10 A% = 23.42
   20 PRINT A%
   RUN
   23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision; i.e. that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision. For example:

   ```
   10 D# = 6#/7
   20 PRINT D#
   RUN
   .8571428571428571
   ```

   The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

   ```
   10 D = 6#/7
   20 PRINT D
   RUN
   .857143
   ```

   The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to +32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded. For example:

   ```
   10 C% = 55.88
   20 PRINT C%
   RUN
   56
   ```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value. For example:

```
10A = 2.04
20 B# = A
30 PRINT A; B#
RUN
 2.04  2.039999961853027
```

2.7 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories. These categories, in order of precedence, are:

1. Functional
2. Arithmetic
3. Relational
4. Logical

2.7.1 Arithmetic Operators

Arithmetic operators of equal priority will be executed from left to right. The operators, in order of precedence, are:

Operator	Operation	Sample Expression
↑	Exponentiation	X↑Y
+,-	Unary plus, minus	-X +3
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)Y$	(X↑2)↑Y
X↑Y↑Z	X↑(Y↑Z)
X(-Y)	X*(-Y) or X*-Y

INTEGER DIVISION AND MODULUS ARITHMETIC

Two additional operators are available in this version of BASIC: Integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to +32767) before the division is performed, and the quotient is truncated to an integer. For example:

10\4 = 2
25.68\6.99 = 3

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)

The precedence of modulus arithmetic is just after integer division.

OVERFLOW AND DIVISION BY ZERO

If during the evaluation of an expression a division by zero is encountered, the "Division by zero" error message is displayed, followed by the line number in which the error occurred. The same thing happens if the evaluation of an exponentiation results in zero being raised to a negative power.

If overflow occurs, the "Overflow" error message is displayed, followed by the line number in which the error occurred. Both errors are intercepted if an ON ERROR GOTO... statement is given.

2.7.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison may then be used to make a decision regarding program flow.

Operator	Relation Tested	Expression
=	Equality	X=Y
<> ><	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y
>= or =>	Greater than or equal to	X>=Y

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. The expression X+Y<(T-1)/Z is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD  J <>0 THEN K=K+1
```

The result of a relational expression is true (integer: -1) or false (integer: 0). These results can be used for further calculations. For example:

```
P=(Q=R)*3 gives P = -3, if Q=R
              and P =  0, if Q≠R.
```

2.7.3 Logical Operators

Logical operators perform a bitwise operation on one integer operands (NOT) or on the corresponding bits of two integer operands (all other logical operators). The result of such an operation is an integer value. In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision. For example:

```
IF D < 200 AND F < 4 THEN 80
IF I > 10 OR K < 0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error occurs).

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator maybe used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
- 1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of 16 zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

2.7.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's intrinsic functions are described in Chapter 3.

BASIC also allows "user defined" functions that are written by the programmer. See DEF FN.

2.7.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used within integer relations:

```
=   <>   <   >   <=   >=
    ><           =<   =>
```

The result of such a comparison also is true (= -1) or false (= 0).

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples of true relations:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL" > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "78/12/09" where B$ = "78/12/08"
```

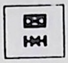
Thus, string comparisons can be used to test string values or to sort strings.

2.8 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the Rubout key.

Rubout has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that you are typing, press:



A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

To delete the entire program that is currently residing in memory, enter the NEW command (see Chapter 3). NEW is usually used to clear memory prior to entering a new program.

For a complete discussion of editing, see the EDIT command in Chapter 3.

3 BASIC COMMANDS AND FUNCTIONS

3.1 INTRODUCTION

All of the BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format:	Shows the correct format for the instruction. See below for format notation.
Purpose:	Tells what the instruction is used for.
Remarks:	Describes in detail how the instruction is used.
Example:	Shows sample programs or program segments that demonstrate the use of the instruction.

3.1.1 Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except square brackets (i.e. commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).
6. Items separated by a vertical bar (|) are mutually exclusive; choose one.

3.1.2 BASIC Functions

The intrinsic functions provided by BASIC are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

c	represents a constant
i and j	represent integer expressions
n	represents a digit
v	represents a variable
x	represents any numeric expression
x$	represents string expressions

Lists are referred to in the text. A list is an item or group of items separated by commas.

If a floating point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

Note - Only integer and single precision results are returned by functions. Double precision functions are not supported.

3.2 ABS

Format: ABS(x)

Purpose: Returns the absolute value of the expression x.

Example:

```
PRINT ABS(7*(-5))
 35
Ok
```

3.3 ASC

Format: ASC(x$)

Purpose: Returns a numerical value that is the ASCII code of the first character of the string x$. (See Appendix E for ASCII codes). If x$ is an empty string, an "Illegal function call" error is returned.

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

3.4. ATN

Format: ATN(x)

Purpose: Returns the arctangent of x (arctangent in radians). The result is in the range -pi/2 to pi/2. The expression x may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN (X)
RUN
? 3
 1.24905
Ok
```

3.5 AUTO

Format: AUTO [i[j]]
where i = line number
and j = increment

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at "i" and increments each subsequent line number by "j". The default for both values is 10. If "i" is followed by a comma but "j" is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will prevent this and generate the next line number. If a text is entered before typing carriage return, the text of the current line is replaced and the next line number is generated.

AUTO is terminated by typing SHIFT-STOP, The line in which SHIFT-STOP is typed is not saved. After SHIFT-STOP is typed, BASIC returns to command level.

Example:

AUTO 100, 50	Generates line numbers 100, 150, 200 ...
AUTO	Generates line numbers 10, 20, 30, 40 ...
AUTO 30	Generates 30, 40, 50...

3.6 CALL

Format: CALL variable name [argument list]

Purpose: To call a machine code subroutine.

Remarks: The CALL statement is one way to transfer program flow to a machine code subroutine (see also the USR function).

"variable name" contains an address that is the entry point in the memory of the subroutine. "variable name" may not be an array variable name. "argument list" contains the arguments that are passed to the subroutine. The arguments may not be expressions but must be variables. Array variables are allowed.

The CALL statement generates the same calling sequence used by BASIC.

See Chapter 5.4, The Use of Machine Code Routines.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
```

This will call the machine code subroutine starting at address -12288 (&HD000).

Note - The statement in line 110 should be executed every time immediately before the statement in 120. Otherwise, the machine code subroutine should save and restore any registers that it uses.

3.7 CDBL

Format: CDBL(x)

Action: Converts "x" to a double precision number.

Example:

```
100 A = 454.67
200 PRINT A; CDBL(A)
RUN
 454.67    454.6700134277344
Ok
```

3.8 CHAIN

Format:

```
CHAIN [MERGE] x$ [,i [,ALL] [,DELETE j [-k]]]
      where x$ = filename
      i = line number or expression
      j and k = line numbers
```

Purpose: To call a program and pass variables to it from the current program.

Remarks: "x$" is the name of the program that is called. "i" is a line number, or an expression that is equivalent to a line number, in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

"i" is not affected by a RENUM command.

3.8.1 The ALL Option

Every variable in the current program is passed to the called program with the ALL option. If it is omitted, the current program must contain a COMMON statement to list the variables that are passed.

3.8.2 The MERGE Option

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. In other words, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file.

Note - If a line number already exists, it is overridden by the called program.

3.8.3 The DELETE Option

If you have brought in an overlay and you want to bring in another, you must delete the old one. Use the DELETE option for this. ("i" and "j" are affected by the RENUM command).

Note - DELETE can only be used with the MERGE option.

Examples:	`CHAIN"PROG1"`	Deletes the calling program, loads and executes "PROG1".
	`CHAIN"PROG2",1000`	Deletes the calling program, loads and executes "PROG2" and continues executing at line 1000 of "PROG2".
	`CHAIN"PROG3",1000,ALL`	Deletes the calling program, loads and executes "PROG3" and continues executing at line 1000 of "PROG3", having passed all variables from the calling program to "PROG3".
	`CHAIN MERGE"PROG4",1000`	The calling program stays in memory, merges "PROG4" and continues executing at line 1000 of the merged program.
	`CHAIN MERGE"PROG5",1000, DELETE 1000 -5000`	Deletes lines 1000-5000 of the calling program, merges "PROG5" and continues executing at line 1000 of the merged program.

Note - If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR or DEFFN statement containing shared variables must be restated in the chained program.

3.9 CHR$

Format: `CHR$(i)`

Action: Returns a string whose one element has ASCII code "i". (ASCII codes are listed in Appendix E.)

CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message.

Example:

```
PRINT CHR$(66)
B
Ok
```

3.10 CINT

Format: `CINT(x)`

Action: Converts "x" to an integer by rounding the fractional portion. If x is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

3.11 CLEAR

Format: `CLEAR [, [expression-1] [, expression-2]]`

Purpose: To undo all declarations of variables and to make a redivision of available memory.

Remarks: "expression-1" specifies the highest memory address to be used by BASIC. Beyond this boundary, applications can place machine code programs or any other information (for example, by POKE statements).

"expression-2" is the size of the stack area to be used by the BASIC system. The minimum value is about 100 but should be specified greater if the application has nested constructions, like FOR...NEXT loops, recursive subroutines or complex expressions. (The recommended size is greater than 400 - the default is about 500).

Example:

```
CLEAR
CLEAR ,&HBFFF,400
```

3.12 CLOSE

Format: CLOSE [[#] file number [,[#] file number...]]

Purpose: To conclude I/O from and to a disk file.

Remarks: "file number" is the number under which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and "file number" finishes upon execution of a CLOSE. The file may then be re-opened using the same or a different file number. Likewise, that file number may then be re-used to open any file.

A CLOSE for a sequential output file writes the final buffer of output.

Note - A normal program end, the NEW command and any program change close all opened disk files automatically. (STOP does not).

3.13 COMMON

Format: COMMON variable list

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program and do not need to be executed.

Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
```

3.14 CONT

Format: CONT

Purpose: To continue program execution after a SHIFT STOP has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the INPUT statement again.

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number.

CONT is invalid if the program has been edited during the break. The program then reports "Can't continue".

3.15 COS

Format: COS(x)

Action: Returns the cosine of x (in radians). The calculation of COS(x) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

Note - To calculate and convert degrees into radians, the following construction can be used.

```
10 PI = 4 * ATN(1)
20 RAD = 180 / PI
30 LPRINT PI
40 LPRINT RAD
50 LPRINT " 1.5 RADIANS = ";1.5 * RAD;"DEGREES"
60 LPRINT " 45  DEGREES = ";45  / RAD;"RADIANS"
RUN
 3.14159
 57.2958
 1.5 RADIANS = 85.9437 DEGREES
 45  DEGREES = .785398 RADIANS
```

3.16 CSNG

Format: `CSNG(x)`

Action: Converts "x" to a single precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

3.17 CVI, CVS, CVD

Format: CVI (2-byte string)
CVS (4-byte string)
CVD (8-byte string)

Purpose: To convert string values to numeric values.

Remarks: Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

See also MKI$, MKS$, MKD$.

3.18 DATA

Format: DATA list of constants

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s) (See READ).

Remarks: DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

"list of constants" may contain numeric constants in any format; i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list). String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed. Note, however, that string constants cannot contain double quotation marks.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement. If the program tries to access DATA via the READ statement after the last DATA item, the program reports an "Out of DATA" error.

DATA statements may be reread from the beginning or from a certain line by use of the RESTORE statement.

Example: See examples in the READ section.

3.19 DEF FN

Format: DEF FN name [parameter list] = function definition

Purpose: To define and name a function that is written by the user.

Remarks: "name" must be a legal variable name. This name, preceded by FN, becomes the name of the function. "parameter list" is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. "function definition" is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The function definition may contain other functions.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

In BASIC, user-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
410 DEF FNAB (X,Y)=X↑3/Y↑2
420 T=FNAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

3.20 DEFINT/SNG/DBL/STR

Format:

```
DEF INT letter range
DEF SNG letter range
DEF DBL letter range
DEF STR letter range
    where letter range = letter/letter-letter
```

Purpose: To declare variable types as integer, single precision, double precision or string, respectively.

Remarks:

Each of these statements declares that all the variable names beginning with one of the specified letters will be of that type. However, a type declaration character always takes precedence over a DEF statement in the typing of a variable.

If no type declaration statements are met, BASIC assumes all variables without declaration characters are single precision variables.

Examples:

`10 DEFDBL L-P`	All variables beginning with the letters L, M, N, O and P will be double precision variables.
`10 DEFSTR A`	All variables beginning with the letter A will be string variables.
`10 DEFINT I-N, B, W-Z`	All variables beginning with the letters I, J, K, L, M, N, B, W, X, Y, Z will be integer variables.

3.21 DEF USR

Format:

```
DEF USR [n] = i
          where n = a digit in the range 0-9
          i = an address
```

Purpose: To specify the entry point of a machine code subroutine.

Remarks: The digit n corresponds to the number of the USR routine whose address is being specified. If "n" is omitted, DEF USR0 is assumed. The value of "i" is the entry point of the USR routine (see Appendix C).

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
.
.
.
200 DEF USR0=24000
210 X=USR0 (Y 2/2.89)
.
.
.
```

3.22 DELETE

Format: DELETE cl [c2]
where cl = first line number to be deleted
c2 = last line number to be deleted

Purpose: To delete program lines.

Remarks: BASIC always returns to command level after a DELETE is executed. If "cl" or "c2" does not exist, an "Illegal function call" error occurs. Each of the line numbers may be replaced by a "." to represent the current line. The current line is the last inserted, listed or updated line. For security reasons, BASIC requires the existence of the first and the last line number. This precaution could prevent, for example, more lines being accidentally deleted after a typing error.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

3.23 DIM

Format: DIM list of array definitions

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0.

The DIM statement sets all the elements of the specified arrays to an initial value of zero in the case of numeric arrays and to an empty string in the case of string arrays.

Note - If no DIM statement has been executed before, the first reference to an array element is an implicit declaration of the array. The number of dimensions is fixed by the number of subscripts. The size of each dimension is made 11 (0-10).

After an array declaration (implicit or explicit), the DIM statement may not be given anymore for this variable name. It will cause a "Duplicate Definition" error message.

Example:

```
10 DIM A (20), B (2,25,3)
20 FOR I=0 TO 20
30 READ A (I)
40 NEXT I
    .
    .
    .
```

3.24 EDIT

Format: EDIT c
where c = line number

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

"c" could be "." to represent the current line. The current line is the last inserted, listed or updated line.

3.24.1 Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and Restarting Edit Mode

Note - In the descriptions that follow, "ch" represents any character, "text" represents a string of characters of arbitrary length and [i] represents an optional integer (the default is 1).

MOVING THE CURSOR

Space	Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.
Rubout	In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

INSERTING TEXT

I	I"text"$ inserts "text" at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press the CODE key, thus allowing you to escape from the insert mode. If Carriage Return is typed during an Insert command, the effect is the same as pressing CODE and then Carriage Return. During an Insert command, the Rubout or Delete key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell is rung and the character is not printed.
X	The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you have finished extending the line, press the CODE key or Carriage Return.

DELETING TEXT

D	[i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the cursor's right, iD deletes the rest of the line.
H	H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

FINDING TEXT

S	The subcommand [i]S"ch" searches for the ith occurrence of "ch" and positions the cursor before it. The character at the current cursor position is not included in the search. If "ch" is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K	The subcommand [i]K"ch" deletes all characters until the ["i"th occurrence of] "ch".The cursor is positioned before "ch", and the deleted characters are enclosed in backslashes.

REPLACING TEXT

C	The subcommand C"ch" changes the next character to "ch". If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

ENDING AND RESTARTING EDIT MODE

"cr"	Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.
E	The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.
Q	The Q subcommand returns to BASIC command level, without saving any of the changes that were made to the line during Edit Mode.
L	The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.
A	The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

Note 1 - If BASIC receives an unrecognizable command or illegal character while in Edit Mode, a bell is rung and the command or character is ignored.

Note 2 - You can exit from the Insert Mode by pressing the CODE key and from any Edit Mode by pressing the Carriage Return key.

3.24.2 Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and press Carriage Return (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

Note 1 - If you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line).

Note 2 - On entering a line, by pressing the Shift key and 9 key (from the numeric pad) together, you can go back and edit the start of the line. If you are in command mode, pressing these two keys recalls the last-entered line.

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of BASIC error codes and messages, see Appendix C.

3.25 END

Format: END

Purpose: To terminate program execution and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

3.26 EOF

Format: EOF file number

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end of file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

3.27 ERASE

Format: ERASE list of array variables

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously-allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Duplicate definition" error occurs.

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

3.28 ERR AND ERL

When an error handling subroutine is entered, the pseudo-variable ERR contains the error code for the error, and the pseudo-variable ERL contains the line number of the line in which the error was detected. (The term "pseudo" is used here because ERR and ERL are variables whose values are assigned by the system only). ERR and ERL are usually used in IF... THEN statements to direct program flow in an error-trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...

```
IF ERR =  error code THEN ...

IF ERL =  line number THEN ...
```

If the line number is not on the right side of the relational operator or is not a constant, it cannot be renumbered by the RENUM command. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

BASIC's error codes are listed in Appendix C.

3.29 ERROR

Format: ERROR integer expression

Purpose:
1) To simulate the occurrence of a BASIC error.
2) To allow error codes to be defined by the user.

Remarks: The value of "integer expression" must be greater than 0 and less than 256. If the value of "integer expression" equals an error code already in use by BASIC (see Appendix C), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed (see Example 1).

To define your own error code, use a value that is greater than any used by BASIC's codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC). This user-defined error code may then be conveniently handled in an error-trap routine (see Example 2).

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message "Unprintable error". Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15                  (you type this line)
String too long           (BASIC types this line)
Ok
```

Example 2:

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B > 5000 THEN ERROR 210
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
.
.
.
```

3.30 EXP

Format: EXP(x)

Action: Returns e to the power of x. x must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

3.31 FIELD

Format: FIELD [#] i1, i2 AS field name

Purpose: To partition records, in random access file fields, so that they can be accessed by name.

Remarks: Before you can get data out of a random buffer after a GET, and before you enter data prior to a PUT, a FIELD statement must be executed.

"i1" is the number under which the file was opened. "i2" is the number of characters to be allocated to "field name". For example:

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file record to the field name N$, the next 10 positions to ID$ and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer. (See LSET/RSET and GET).

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened, otherwise, a "Field overflow" error occurs. (The default record length is 256).

Any number of FIELD statements may be executed for the same file. Subsequent FIELD statements do not affect former ones, except for those field names which occur in both former and subsequent statements.

Note - Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer will be changed and point to string spaces in memory.

3.32 FIX

Format: FIX(x)

Action: Returns the truncated integer part of "x". FIX(x) is equivalent to SGN(x)*INT(ABS(x)). The difference between FIX and INT is that FIX does not return the next lower number for negative "x".

Examples:

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

3.33 FOR ... NEXT

Format:

```
FOR vl = x TO y [STEP z]
.
.
.
NEXT [v2] [,v3...]
     where x, y and z are numeric expressions.
```

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: vl, 2 and 3 are numeric variables used as counters. However, they are not array elements. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value. However, the body of the loop is executed at least one time.

3.33.1 Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them. In this case, the variable names may not be omitted but must be mentioned in reverse sequence (see Example 4).

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is met before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
1   20
3   30
5   40
7   50
9   60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
RUN
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

Here, the loop executes ten times. The final value for the loop variable is always set after the initial value is set.

Example 4:

```
10 FOR I=0 TO 1
20 FOR J=0 TO 1
30 PRINT I+J
40 NEXT J,I
```

This is an example of nested FOR...NEXT loops with a common end.

3.34 FRE

Format:

```
FRE(0)
FRE("")
```

Action: Arguments to FRE are dummy arguments.

FRE(0) returns the number of bytes in memory that are not being used by BASIC but are under the control of BASIC.

FRE("") forces a garbage collection before returning the number of free bytes. BASIC will not initiate this garbage collection until all free memory has been used up.

Examples:

```
PRINT FRE(0)
 20838
Ok
```

```
PRINT FRE("")
 50
Ok
```

3.35 GET

Format: GET [#] file number [, record number]

Purpose: To read a record from a random disk file into a random buffer.

Remarks: "file number" is the number under which the file was opened. If "record number" is omitted, the next record (after the last GET) is read into the buffer. The lowest possible record number is 1.

3.36 GOSUB

Format:

```
GOSUB c
    where c = line number
```

Purpose: To invoke a subroutine.

Remarks: "c" is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. If "c" is not valid, an "Undefined line number in xx" error is reported.

Example 1:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE IN PROGRESS"
50 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

Example 2:

```
                     1st. Invoke:   2nd. Invoke:

10 GOSUB 30            GOSUB 50       ?"LINE 50",
                       ?"LINE 40",    ?"LINE 40",
                       ?"LINE 50",    ?"LINE 50",
20 END                 END            END
30 GOSUB 50
40 ?"LINE 40",
50 ?"LINE 50",
60 RETURN
RUN
LINE 50    LINE 40    LINE 50
Ok
```

3.37 GOTO

Format: GOTO c
where c = line number

Purpose: To branch unconditionally into another program sequence starting at the specified line number.

Remarks: If "c" is an executable statement, that statement and those following are executed. If it is a non-executable statement, execution proceeds at the first executable statement met after "c". If "c" is not valid, an "undefined line number in xx" error is reported.

Example:

```
10 READ R
20 PRINT "R=";R,
30 A = 3.14*R↑2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5, 7, 12
Ok
RUN
R = 5              AREA = 78.5
R = 7              AREA = 153.86
R = 12             AREA = 452.16
Out of DATA in 10
Ok
```

3.38 HEX$

Format: HEX$(x)

Action: Returns a string which represents the hexadecimal value of the decimal value of the decimal argument. "x" is rounded to an integer before HEX$(x) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

3.39 IF...THEN...ELSE, IF...GOTO

Format:

```
IF expression THEN statement.. | line number
[ELSE statement.. | line number]
```

OR

```
IF expression GOTO line number
[ELSE statement.. | line number]
```

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of "expression" is not zero, the THEN or GOTO clause is executed. ("true" is represented by a non-zero value). THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of "expression" is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

3.39.1 Nesting of IF Statements

IF ...THEN...ELSE ... statements may be nested. Nesting is limited only by the length of the line. For example:

```
IF X > Y THEN PRINT "GREATER" ELSE IF Y > X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A <> C"
```

will not print "A <> C", when A <> B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

Note - When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0) < 1.0E-6 THEN ...
```

This test returns true if the value of A is in the range 0.999999 - 1.000001.

Example 1:

```
200 IF I THEN GOTO 300
```

Program flow goes to line number 300 if I is not zero; otherwise, the program continues at the next line after 300.

Example 2:

```
100 IF (I < 20) AND (I > 10) THEN DB= 1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
.
.
.
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

3.40 INKEY$

Format: INKEY$

Purpose: To test whether a key has been pressed or not.

Remarks: INKEY$ returns an empty string if no key was pressed. Otherwise a string of length 1 is given which contains the character represented by the keytop.

Example:

```
100 A$=INKEY$: IF A$="" THEN 100
110 REM A$ NOW CONTAINS THE VALUE OF THE KEY THAT WAS PRESSED
```

3.41 INP

Format: INP(i)

Action: Returns the byte read from port "i". "i" must be in the range 0 to 255.

Example:

```
100 A=INP(255)
```

3.42 INPUT

Format: INPUT [prompt string;] list of variables

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If "prompt string" is included, the string is printed before the question mark. The required data is then entered at the terminal.

The data that is entered is assigned to the variable(s) given in "variable list". The number of data items supplied should not exceed the number of variables in the list. If the number is smaller, a prompt with ?? should be done for the rest. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks. However, if the string is surrounded by quotation marks, the quotation marks are stripped off).

Responding to INPUT with carriage-return (i.e. without entering values) leaves the contents of the rest of the variables unchanged.

INPUT is illegal in the direct mode.

Examples:

```
10  INPUT X
20  PRINT X "SQUARED IS" X↑2
30  END
RUN
?  5      (The 5 was typed in by the user
           in response to the question mark.)
 5  SQUARED IS 25
Ok


LIST
10 PI= 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R↑2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS ? 7.4     (User types 7.4)
THE AREA OF THE CIRCLE IS  171.946

WHAT IS THE RADIUS ?
etc.

WHAT IS THE RADIUS ?
etc.
```

3.43 INPUT#

Format: INPUT# file number, variable list

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: "file number" is the number used when the file was opened for input. "variable list" contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name). With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. When assigning numeric variables, the string read from a file is evaluated as the argument of the function VAL. The first character that is not a space, carriage return or line feed is assumed to be the start of a number. The number finishes on a space, carriage return, line feed or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character met that is not a space, carriage return or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string and will finish on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

3.44 INPUT$

Format: INPUT$(X[,[#]Y])

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except SHIFT-STOP, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5  'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
    HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

3.45 INSTR

Format: `INSTR([i,]x$,y$)`

Action: Searches for the first occurrence of string y$ in x$ and returns the position at which the match is found. Optional offset i sets the position for starting the search. i must be in the range 0 to 255. If i LEN(x$) or if x$ is an empty string or if y$ cannot be found, INSTR returns 0. If only y$ is an empty string, INSTR returns i, or 1 (if i is omitted).

Example:

```
10 X$  =  "ABCDEB"
20 Y$  =  "B"
30 PRINT INSTR(X$,Y$); INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

3.46 INT

Format: `INT(x)`

Action: Returns the largest integer<=x.

Examples:

```
PRINT INT (99.89)
  99
Ok

PRINT INT(-12.11)
-13
Ok
```

3.47 KILL

Format: KILL file name

Purpose: To delete a file from a disk.

Remarks: If a KILL command is given for a file that is open, a "File already open" error occurs. KILL is used for all types of disk files: program files, random data files and sequential data files.

Note - For security reasons, the KILL command requires the full file name; i.e. the file name and file extension.

Example:

```
200 KILL "MYJOB.BAS"
```

3.48 LEFT$

Format: LEFT$(x$,i)

Action: Returns a string comprised of the leftmost "i" characters of x$. "i" must be in the range 0 to 255. If "i" is greater than LEN (x$), the entire string (x$) will be returned. If "i" = 0, an empty string is returned.

Example:

```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
RUN
30 PRINT B$
BASIC
Ok
```

3.49 LEN

Format: LEN(x$)

Action: Returns the number of characters in x$. Non-printable characters and blanks are counted.

Example:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
Ok
```

3.50 LET

Format: [LET] variable = expression

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional; i.e. the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110  LET D=12
120  LET E=12↑2
130  LET F=12↑4
140  LET SUM=D+E+F
  .
  .
  .
```

or

```
110 D=12
120 E=12↑2
130 F=12↑4
140 SUM=D+E+F
  .
  .
  .
```

3.51 LINE INPUT

Format: LINE INPUT [prompt string;] string variable

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to "string variable".

A LINE INPUT may be escaped by typing SHIFT-STOP. BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT. LINE INPUT is illegal in the direct mode.

Responding to LINE INPUT with a Carriage Return (i.e. without entering a value) makes the contents of the variable an empty string.

3.52 LINE INPUT#

Format: LINE INPUT# file number, string variable

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: "file number" is the number under which the file was opened. "string variable" is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is met, it is preserved).

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION?";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES 234,4 MEMPHIS
LINDA JONES 234,4 MEMPHIS
Ok
```

3.53 LIST

Format: LIST [c1] [-] [c2]
where c1 = first line number
c2 = last line number

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC always returns to command level after a LIST is executed.

Listing is terminated either by the end of the program or by typing SHIFT STOP.

The format allows the following options:

1. LIST
   or
   LIST -
   All of the program currently in the memory is listed at the terminal.

2. LIST "c1"
   Only the specified line number is listed.

3. LIST "c1" -
   The program is listed beginning at that line.

4. LIST - "c2"
   All lines from the beginning of the program through that line are listed.

5. LIST "c1" - "c2"
   The entire range is listed.

Remember that "." represents the current line number.

Examples:

LIST	Lists the program currently in memory.
LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000 inclusive.
LIST 500	Lists line 500.
LIST .	Lists line 500 again.
LIST .-	Lists the program from line 500 to the end.
LIST .	Now lists the last line (the last line listed has become the current line).

3.54 LLIST

Format:

```
LLIST [cl] [-] [c2]
      where cl = first line number
            c2 = last line number
```

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes an 80-character wide printer (see WIDTH).

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST.

Example: See the examples for LIST.

3.55 LOAD

Format: LOAD file name [,R]

Purpose: To load a file from disk into memory.

Remarks: "file name" is the name that was used when the file was SAVEd (the default extension .BAS is supplied).

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open.

Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example:

```
LOAD "MYJOB",R
```

3.56 LOC

Format: LOC (file number)

Action: With random disk files, LOC returns the next record number to be used if a GET or PUT (without a record number) is executed. With sequential files, LOC returns the number of sectors (128-byte blocks) read from or written to the file since it was opened.

Example:

```
200 IF LOC(1) > 50 THEN STOP
```

3.57 LOG

Format: `LOG(x)`

Action: Returns the natural logarithm of x. x must be greater than zero.

Example:

```
PRINT LOG(45/7)
 1.86075
```

3.58 LPOS

Format: LPOS(x)

Action: Returns the current position of the line printer print head within the line printer buffer. It does not necessarily give the physical position of the print head.

"x" is a dummy argument.

Example:

```
100 IF LPOS (X) > 60 THEN LPRINT CHR$(13)
```

3.59 LPRINT AND LPRINT USING

Format: LPRINT [list of expressions]

LPRINT USING string exp; list of expressions

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer.

LPRINT assumes the default of an 80-character print width (see WIDTH).

3.60 LSET AND RSET

Format: LSET string variable = string expression
RSET string variable = string expression

Purpose: To move data from memory to a random·file buffer (in preparation for a PUT statement).

Remarks: If "string expression" requires fewer bytes than were FIELDed to "string variable", LSET left-justifies the string in the field and RSET right-justifies the string. (Spaces are used to pad the extra positions). If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they can be LSET or RSET. See the MKI$, MKS$ and MKD$ functions.

Example:

```
150 LSET A$=MKS$(AMT)
160 RSET D$=DESC($)
```

Note - LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be useful for formatting printed output.

3.61 MERGE

Format: MERGE filename

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: "filename" is the name used when the file was SAVEd (the default extension .BAS is supplied). The file must have been SAVEd in ASCII format (if not, a "Bad file mode" error occurs).

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (Merging may be thought of as "inserting" the program lines on disk into the program in memory).

BASIC always returns to command level after executing a MERGE command.

Example:

```
MERGE "MYJOB"
```

3.62 MID$

Format: MID$ (string expl, n [,m]) = string exp2
where n and m are integer expressions and "string expl" and "string exp2" are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in "string expl", beginning at position n, are replaced by the characters in "string exp2". The optional m refers to the number of characters from "string exp2" that will be used in the replacement. If m is omitted, all of "string exp2" is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of "string expl".

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ may also be used as a function that returns a substring of a given string.

3.63 MID$

Format: MID$(x$,i[,j])
where i = first character position of substring of x$.
j = length of substring.

Action: Returns a string of length "j" characters from x$ beginning with the "i"th character. If "j" is omitted or if there are fewer than "j" characters to the right of the "i"th character all rightmost characters beginning with the "i"th character are returned.

If i > LEN(X$), MID$ returns a null string.

Example:

```
LIST
10 A$="GOOD"
20 B$=" MORNING EVENING AFTERNOON"
30 PRINT A$; MID$(B$,9,8)
Ok
RUN
GOOD EVENING
Ok
```

3.64 MKI$, MKS$, MKD$

Format:

```
MKI$ (integer expression)
MKS$ (single precision expression)
MKD$ (double precision expression)
```

Action: Numeric values are converted to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement, must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
100 AMT=(K+T)
110 FIELD #1. 8 AS D$, 20 AS N$
120 LSET D$ = MKS$(AMT)
130 LSET N$ = A$
140 PUT #1
.
.
.
```

3.65 NAME

Format: NAME old-filename AS new-filename

Purpose: To change the name of a disk file.

Remarks: "old-filename" must exist and "new-filename" must not exist, otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space and with the new name.

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was called ACCTS will now be named LEDGER.

3.66 NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

Note - The NEW command does not reset the specifics of the CLEAR command.

3.67 OCT$

Format: OCT$(i)

Action: Returns a string which represents the octal value of the decimal argument. "i" is rounded to an integer before OCT$(i) is evaluated.

Note: -32768 < "i" < 65536

Example:

```
PRINT OCT$(24)
 30
Ok
```

3.68 ON ERROR GOTO

Format: ON ERROR GOTO i
where i = line number

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error-trapping has been enabled all errors detected, including direct mode errors (e.g. syntax errors), will cause a jump to the specified error-handling subroutine. If "i" does not exist, an "Undefined line" error results. To disable error-trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error-trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note - If an error occurs during execution of an error-handling subroutine, the BASIC error message is printed and execution terminates. Error-trapping does not occur within the error-handling subroutine.

To return from the error-handling subroutine, the RESUME command must be used.

Example:

```
10   ON ERROR GOTO 1000
.
.
.
1000 IF ERR=11 THEN RESUME 520
1010 ON ERROR GOTO 0
```

(Lines 520 onwards contain a "Division by zero" error-handling routine).

3.69 ON...GOSUB AND ON...GOTO

Format: ON expression GOTO list of line numbers

ON expression GOSUB list of line numbers

Purpose: ON expression GOTO branches to a specified line number, depending on the value of "expression".

ON expression GOSUB invokes a subroutine starting at the specified line number, depending on the value of "expression".

Remarks: The value of "expression" determines the position in the list of the line number which will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

If the value of "expression" is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of "expression" is negative or greater than 255, an "Illegal function call" error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

3.70 OPEN

Format:

```
OPEN mode, [#] i, x$, [j]
      where i = file number
           x$ = file name
            j = record length
```

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

"mode" is a string expression whose first character is one of the following:

O specifies sequential output mode
I specifies sequential input mode
R specifies random input/output mode

"i" is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is open and is used to refer other disk I/O statements to the file. (It is recommended that the number be the maximum that the system bootstrap message allocates - "How many files? (0-15, default 3")).

"x$" is a string expression containing a name that conforms to the BASIC rules for file names.

"j" is an integer expression which, if included, sets the record length for random files. The default record length is 256 bytes. A syntax error will occur if "j" is specified when a sequential file is opened.

Note - A file can be OPENed for sequential input or random access on more than one file number at a time. But a file may be OPENed for output on only one file number at a time.

Example:

```
10 OPEN "I",2,"INVEN"
```

3.71 OPTION BASE

Format: OPTION BASE 0|1

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement "OPTION BASE 1" is executed, the lowest value an array subscript may have is 1. If more than one OPTION BASE statement is specified in a program, a "Duplicate definition" error occurs until a CLEAR command is given.

Note - The first use of an array implicitly sets the option base equal to zero if not already specified.

3.72 OUT

Format: OUT i,j
where i and j are integers in the range 0 to 255.

Purpose: To send a byte to a machine output port.

Remarks: "i" is the port number, and the integer "j" is the data to be transmitted.

Example:

```
100 OUT 32, 100
```

3.73 PEEK

Format: PEEK(i)

Action: Returns the byte (integer in the range 0 to 255) read from memory location "i". "i" must be in the range -32768 to 65535.

Example:

```
A=PEEK(&H5A00)
```

3.74 POKE

Format:

```
POKE i, j
     where i = address
     j = value
```

Purpose: To write a byte into a memory location.

Remarks: The integer expression "i" is the address of the memory location to be POKEd. The integer expression "j" is the data to be POKEd. "j" must be in the range 0 to 255. "i" must be in the range 0 to 65535 but may also be given in the range -32768 to -1 (corresponding with addresses +32768 to +65535).

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

POKE and PEEK are useful for efficient data storage and loading machine code information for the next program.

Example:

```
10 POKE &H5A00, &HFF
```

The hexadecimal range is from &H0000 to &HFFFF.

Hexadecimal		Decimal
&H0000-&H7FFF	=	0:32767
&H8000-&HFFFF	=	-32768:-1

For example, a simple program to pass information to the next program:

Program 1

```
10  CLEAR 50, &HDFFF
      .
      .
      .
50  INPUT A%
60  POKE &HE000, A%
```

Program 2

```
100 A%=PEEK(&HE000)
```

3.75 POS

Format: `POS(x)`

Action: Returns the current cursor position in the current line. The leftmost position is 1. "x" is a dummy argument.

Example:

```
IF POS(X) < 60 THEN PRINT CHR$(13)
```

3.76 PRINT

Format: PRINT [list of expressions]

Purpose: To output data at the terminal.

Remarks: If "list of expressions" is omitted, a blank line is printed. If "list of expressions" is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks).

3.76.1 Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 positions each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return linefeed is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Single precision numbers in the range 0.01 to 999999 are represented normally. Double precision numbers are represented normally up to 16 figures; i.e. 9--------------9. Outside these boundaries, numbers are represented in floating point notation. If you wish to avoid this, you must employ the PRINT USING command.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*-5, X↑5
30 END
RUN
 10            0             -25           3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X*X "AND";
30 PRINT X "CUBED IS" X*X*X
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space). In line 40, a question mark is used instead of the word PRINT.

3.77 PRINT USING

Format: PRINT USING x$; list of expressions

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: "list of expressions" is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. "x$" is a string literal (or variable) that is comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

3.77.1 String Fields

When PRINT USING is used to print strings, one of two formatting characters may be used to format the string field:

!	Specifies that only the first character in the given string is to be printed.
n spaces	Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "'  '";A$;B$
50 PRINT USING "'   '";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

Note - Backslashes are printed as apostrophes.

3.77.2 Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

#	A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.
.	A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##    ";10.2,5.3,66.789,.234
10.20       5.30    66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+	A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95     +2.40    + 55.60     - 0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-     22.45       7.01-
```

****** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4     *-0.9       765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
   $456.78
```

****$** The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING  "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (↑↑↑↑) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

↑↑↑↑

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##↑↑↑↑"; 234.56
  2.35E+02
```

```
PRINT USING ".####↑↑↑↑-";888888
  .8889E+06
```

```
PRINT USING "+.##↑↑↑↑";123
+.12E+03
```

%

If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

3.78 PRINT# AND PRINT# USING

Format: PRINT# file number, [USING string expression;]
list of expressions

Purpose: To write data to a sequential disk file.

Remarks: "file number" is the number used when the file was opened for output. "string expression" is comprised of formatting characters, as described in the previous section, "PRINT USING". The expressions in "list of expressions" are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In "list of expressions", numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk).

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in "list of expressions".

Examples: Let A$="CAMERA" and B$="93604-1". The statement:

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is CAMERA,93604-1 which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns or line feeds, write them to disk surrounded by explicit quotation marks.

Let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement:

```
PRINT#1,A$;B$
```

would write CAMERA, AUTOMATIC 93604-1 to disk and the statement:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34).

The statement:

```
PRINT#1, CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""    93604-1"
```

and the statement:

```
INPUT#1.A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and "   94603-1" to B$.

The PRINT statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

3.79 PUT

Format: PUT [#] file number [, record number]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: "file number" is the number under which the file was opened. If "record number" is omitted, the record will have the next available record number (after the last PUT). The lowest possible record number is 1.

3.80 RANDOMIZE

Format: RANDOMIZE (expression)

Purpose: To reseed the random number generator.

Remarks: If "expression" is omitted, BASIC suspends program execution and, before executing RANDOMIZE, asks for a value: "Random number seed (-32768 to 32767)?"

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

Example:

```
10 RANDOMIZE
20 FOR I=1 to 5
30 PRINT RND;
40 NEXT I
RUN
Random seed number (-32768 to 32767)?
3          (You type in "3")
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random seed number (-32768 to 32767)?
4          (You type in "4")
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random seed number (-32768 to 32767)?
3          (Same sequence as first run)
 .88598  .484668  .586328  .119426  .709225
Ok
```

Note - The following construction should be used to generate a really random start value:

```
10 RANDOMIZE 256 * PEEK(&H6011) + PEEK(&H6010) - 32768
```

3.81 READ

Format: READ list of variables

Purpose: To read values from a DATA statement and assign them to variables.

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (the values will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in "list of variables" exceeds the number of elements in the DATA statement(s), an "Out of DATA" message is printed. If the number of variables specified is fewer than the number of elements in the DATA statment(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start or from a certain line, use the RESTORE statement.

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$, S$, Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$, S$, Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO        80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

3.82 REM

Format: REM remark

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Example:

```
.
.
.
120  REM CALCULATE AVERAGE VELOCITY
130  FOR I=1 TO 20
140  SUM=SUM + V (I)
150  NEXT
160  AV=SUM/120
.
.
.
```

3.83 RENUM

Format:

```
RENUM [[cl] [, [c2] [, increment]]]
      where cl = new line number
            c2 = old line number
```

Purpose: To renumber program lines.

Remarks: "cl" is the first line number to be used in the new sequence. The default is 10. "c2" is the line in the current program where renumbering is to begin. The default is the first line of the program. "increment" is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a non-existent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

Note - RENUM cannot be used to change the order of program lines (e.g. RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will be incremented by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will be incremented by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so that they start with number 1000 and are incremented by 20.

3.84 RESET

Format: RESET

Purpose: To inform the system that a new disk has been inserted or to reset disk errors.

Remarks: The system regards all files as closed. The disk administration of Disk 1 is read in if you are working under SYSTEM A. Both disk administrations are read in with SYSTEM B.

The system cannot tell whether a disk has been changed. To prevent writing in the directory of the wrong disk, use the RESET command in your program at all logical places where a disk can be changed. For security reasons, the system forbids you to write to a disk after any error during access until a RESET command has been given, either as a program statement or as a direct command.

Example (How to program a disk change):

```
1100 CLOSE#1:CLOSE#3     :REM CLOSE ALL DATASETS ON
1110                     :REM DISK IN DRIVE 2
1120 INPUT "REPLACE DISK IN DRIVE 2 BY DISK 402A,
     THEN PRESS: ENTER";A$
1130 RESET:
```

Note - Make sure that you close all (output) files on a disk before you remove it, to prevent parts of your files being lost.

3.85 RESTORE

Format: RESTORE [c]
where c = line number

Purpose: To allow DATA statements to be reread from a specified point.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If "c" is specified, the next READ statement accesses the first item in the first DATA statement after that line.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
.
.
.
```

3.86 RESUME

Formats:

```
RESUME

RESUME 0

RESUME NEXT

RESUME c
    where c = line number
```

Purpose: To continue program execution after an error-recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME c	Execution resumes at "c".

A RESUME statement causes a "RESUME without error" message to be printed, if no error trap routine is specified.

Example:

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230)AND(ERL=90)THEN PRINT "TRY
AGAIN": RESUME 80
.
.
.
```

3.87 RETURN

Format: RETURN

Purpose: To return from a subroutine.

Remarks: Program flow continues after the last unmatched GOSUB statement.

Example: See GOSUB.

3.88 RIGHT$

Format: RIGHT$(x$,i)

Action: Returns the rightmost "i" characters of string "x$". If "i" LEN(x$), "x$" is returned. If "i" = 0, an empty string is returned.

Example:

```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

3.89 RND

Format: RND (x)

Action: Returns a random number between 0 and 1. However, "x" $<$ 0 always restarts the same sequence for any given "x". For example, if RND is called with a negative argument at the beginning of the program, the same sequence of random numbers, is generated each time the program is RUN.

"x" $>$ 0 generates the next random number in the sequence. "x" = 0 repeats the last number generated.

Example:

```
10   X = RND(-1)
20   X = 1
30   FOR I = 1 TO 5
40   PRINT INT(RND(X)*100);
50   NEXT
RUN
 49  67  98  73  78
```

3.90 RUN

Format: RUN [i]
where i = line number

Purpose: To execute the program currently in memory.

Remarks: If "i" is specified, execution begins on that line. Otherwise, execution begins at the first line number.

Note - RUN clears all variables; GOTO does not.

Example:

```
RUN
RUN 10
GOTO 10
```

3.91 RUN

Format: `RUN filename [,R]`

Purpose: To load a file from disk into memory and access it.

Remarks: "filename" is the name used when the file was SAVEd. (The default extension .BAS is supplied).

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain open.

Example:

```
RUN "NEWFIL",R
```

3.92 SAVE

Format: `SAVE filename [,A|,P]`

Purpose: To save a program file on disk.

Remarks: "filename" is a quoted string that conforms to the BASIC requirements for filenames. (The default extension .BAS is supplied). If "filename" already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:

```
SAVE"COM2",A
SAVE"PROG",P
```

3.93 SGN

Format: SGN(x)

Action: If x > 0, SGN(x) returns 1.
If x = 0, SGN(x) returns 0.
If x < 0, SGN(x) returns -1.

Example: ON SGN(X)+2 GOTO 100, 200, 300 branches to 100 if x is negative, to 200 if x is 0 and to 300 if x is positive.

3.94 SIN

Format: SIN(x)

Action: Returns the sine of x (in radians). SIN(x) is calculated in single precision.

Example:

```
PRINT SIN (1.5)
 .997495
Ok
```

3.95 SPACE$

Format: SPACE$(i)

Action: Returns a string of spaces of length "i". "i" must be in the range 0 to 255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
 Ok
```

3.96 SPC

Format: SPC(i)

Action: Prints "i" blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. "i" must be in the range 0 to 255.

Example:

```
PRINT "OVER" SPC(15) " THERE"
OVER               THERE
Ok
```

3.97 SQR

Format: SQR(x)

Action: Returns the square root of "x". "x" must be>= 0, otherwise an "Illegal function call" error occurs.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

3.98 STOP

Format: STOP

Purpose: To suspend program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to end execution. When a STOP is met, the following message is printed:

```
Break in nnnnn
```

Note - nnnnn = line number.

BASIC always returns to command level after a STOP is executed. Execution of the next statement is resumed by issuing a CONT command.

Example:

```
10 INPUT A,B,C
20 K=A↑2*5.3:L=B↑3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

Note - Normally, the STOP statement is used temporarily for debugging purposes. During the suspension of program activity, all variables may be inspected and changed. A program change, however, clears all variables and the CONT statement will not be accepted.

3.99 STRING$

Formats:

```
STRING$ (i,j)
STRING$ (i,x$)
```

Action: Return a string of length "i" whose characters all have ASCII code "j" or the first character of "x$". "i" and "j" must be in range 0 to 255.

Example:

```
10 X$ = STRINGS$ (10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
-----------MONTHLY REPORT------------
Ok
```

3.100 STR$

Format:

```
STR$(x)
```

Action: Returns a string representation of the value of "x". This string has the same format as the format of "x". It may print:

- one leading blank or minus sign
- one trailing blank
- if necessary, a single/double precision
- floating point notation

Example:

```
5  REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

3.101 SWAP

Format:

```
SWAP v1,v2
     where v1 = variable-1
           v2 = variable-2
```

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

3.102 TAB

Format: TAB(i)

Action: Spaces to position "i" on the current output line. If "i" > line length, the line length is subtracted from "i" and a "jump" is made to the next line (repeating). If the current print position is already beyond space "i", the function TAB has no effect. Space 1 is the leftmost position, and the rightmost position is the width minus one. "i" must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10  PRINT "NAME" TAB(25) "AMOUNT": PRINT
20  READ A$,B$
30  PRINT A$ TAB(25) B$
40  DATA "G. T. JONES", "$25.00"
RUN
NAME                        AMOUNT

G. T. JONES                 $25.00
Ok
```

3.103 TAN

Format: TAN(x)

Action: Returns the tangent of x (specified in radians). TAN(x) is calculated in single precision. If TAN overflows, an error "Division by zero" occurs, machine infinity with the appropriate sign is supplied as the result and execution continues. (Machine infinity is 1.70141E+38).

Example:

```
 10 Y = Q*TAN(X)/2
```

3.104 TRON/TROFF

Format:

```
TRON

TROFF
```

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
'10''20''30''40' 1  10  20
'50''60''30''40' 2  20  30
'50''60''70'
Ok
TROFF
Ok
```

3.105 USR

Format:

```
USR[n](a)
     where n = a digit in the range 0-9.
```

Action: Calls the user's assembly language subroutine with the argument "a" and returns a value of the same type as "a". "a" may be any type. "n" is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If "n" is omitted, USR0 is assumed. (See the DEF USR section).

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
.
.
.
```

3.106 VAL

Format: `VAL(x$)`

Action: Returns the numerical value of string x$. If the first character of x$ is not +, -, &, or a digit, VAL(x$)=0.

Example:

```
10  PRINT VAL("12ABC")
20  PRINT VAL("-3.5PQR")
30  A$="12"
40  PRINT VAL("&H"+A$)
RUN
 12
-3.5
 18
Ok
```

3.107 VARPTR

Format: VARPTR(variable name)

Action: Returns the address of the first byte of data identified with "variable name". A value must be assigned to "variable name" prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range -32768 to +32767. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR (A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note - All simple variables should be assigned before calling VARPTR for an array element, because the addresses of the arrays change whenever a new simple variable is assigned.

Example:

```
10 REM SEE WARNING
20 DEFINT A-Z
30 DIM X(10)
40 X(8)=31
50 A=VARPTR(X(8))
60 I = 111 : REM CAUSES A SHIFT OF THE ARRAY
70 LPRINT A
80 LPRINT PEEK (A+1);PEEK(A)
90 LPRINT
100 A=VARPTR(X(8))
110 LPRINT A
120 LPRINT PEEK(A+1);PEEK(A)
RUN
-27865
 0  0

-27859
 0  31
```

Note - The printed results depend on your configuration and version of BASIC.

3.108 WAIT

Format: WAIT port number, i[,j]
where i and j are integer expressions

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression "j", and then AND'ed with "i". If the result is zero, BASIC loops back and reads the data at the port again. If the result is non-zero, execution continues with the next statement. If "j" is omitted, it is assumed to be zero.

Note - It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to restart the machine manually.

Example:

```
100 WAIT 32,2
```

3.109 WHILE...WEND

Format: WHILE expression...[loop statements]...WEND

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If "expression" is not equal to zero (i.e. true), "loop statements" are executed until the WEND statement is met. BASIC then returns to the WHILE statement and checks "expression". If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
120        FLIPS=0
130        FOR I=1 TO J=1
140                IF A$(I) A$(I+1) THEN
                         SWAP A$(I),A$(I+1):FLIPS=1
150        NEXT I
160 WEND
```

Note - WHILE...WEND is not implemented in most BASICs.

3.110 WIDTH

Format: WIDTH [LPRINT] i
where i = integer expression

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

"i" must have a value in the range 1 to 255. The default width is 80 characters for printers and screen.

If "i" is 255, the line width is "infinite"; i.e. BASIC never inserts a carriage return. However, the position of the cursor of the print head, as given by the POS or LPOS function, returns to zero after position 255.

3.111 WRITE

Format: WRITE [list of expression]

Purpose: To output data at the terminal.

Remarks: If "list of expressions" is omitted, a blank line is output. If "list of expressions" is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement without leading and trailing blanks.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80,90,"THAT'S ALL"
Ok
```

3.112 WRITE#

Format: WRITE# filenumber, list of expressions

Purpose: To write data to a sequential file

Remarks: "file number" is the number under which the file was opened in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1".

The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1'
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

4 DATA FILE HANDLING

Flexible disks can be used both for saving programs so that they can be run at a later date and also for storing data that is either created or processed by a BASIC program. The beginning of this chapter deals with the general concepts of working with disk files. The remainder of the chapter contains a list of all the BASIC disk handling commands.

4.1 FILE COMMANDS

The BASIC rules regarding variable names also apply for file names. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the default drive is assumed. The default drive can be changed using the SYSTEM command. 'A' refers to disk drive number 1 and 'B' refers to disk drive number 2. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN filename commands if no "." appears in the filename and the filename is less than 9 characters long. For example:

```
SAVE "PROG1"
```

will cause a file called PROG1.BAS to be stored on the disk.

4.1.1 RESET Command

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

4.1.2 LOF Function

Format: LOF (file number)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (64 records), then LOF returns the true length of the file.

Example:

```
110 PRINT "MY CURRENT RECORD NUMBER ON FILE 1 IS", LOF(1)
```

4.1.3 EOF Function

With BASIC the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or any other algorithm.

Disk I/O procedures for the new BASIC user are examined in this section. If you are new to BASIC or if you are getting disk-related errors, read through these procedures and program examples to make sure that you are using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to BASIC's requirements for filenames. The file system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

4.1.4 Program File Commands

Here is a review of the commands and statements used in program file manipulation.

SAVE "filename"[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD "filename"[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and access the same data files.

RUN "filename"[,R]	Loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.
MERGE "filename"	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL "filename"	Deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file.
NAME	To change the name of a disk file, execute the NAME statement; NAME "oldfile" AS "newfile". NAME may be used with program files, random files or sequential files.

4.1.5 Protected Files

If you wish to save a program in an encoded binary format use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. The attempt to list or edit the protected program results in an "Illegal function call".

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.2 SEQUENTIAL FILES

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with seque files are:

```
OPEN   PRINT#          INPUT#         WRITE#
       PRINT# USING    LINE INPUT#

CLOSE  EOF    LOC
```

2.1 Creating a Sequential File

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	`OPEN "O",#1,"DATA"`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	`PRINT#1,A$;B$;C$`
3. To access the data in the file, you must CLOSE the file and re-OPEN it in "I" mode.	`CLOSE#1` `OPEN "I",#1,"DATA"`
4. Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,A$;B$;C$`

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN


NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Program B-1 - Create Sequential Data File

4.2.2 Accessing a Sequential File

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program B-2 - Accessing a Sequential File

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 256-byte block of data.

4.2.3 Adding Data To A Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its contents. Follow the procedure over the page for a file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data that you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO COPY
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

Program B-3 - Adding Data To A Sequential File

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

4.3 RANDOM FILES

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk - it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

There follows a list of statements and functions that are used with random files.

OPEN FIELD LSET/RSET GET

PUT CLOSE LOC

MKI$ CVI
MKS$ CVS
MKD$ CVD

4.3.1 Creating A Random File

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode).
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.
3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.
4. Write the data from the buffer to the disk using the PUT statement.

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Note - Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Program B-4 - Create a Random File

4.3.2 Accessing A Random File

The following program steps are required to access a random file:

1.	OPEN the file in "R" mode.	`OPEN "R",#1,"FILE",32`
2.	Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.	`FIELD #1 20 AS N$ 4 AS A$, 8 AS P$`

Note -In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

   `GET #1,CODE%`

4. The data in the buffer may be acessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the two-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE"
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Program B-5 - Access a Random File

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) = 50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
100 OPEN "R",#1,"INVEN.DAT",39
110 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
120 PRINT:PRINT "FUNCTIONS:":PRINT
130 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
190 PRINT 7,"END"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION < 1) OR (FUNCTION >7) THEN PRINT
    "BAD FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680,970
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$) < >255 THEN INPUT"OVERWRITE";A$:
    IF A$ < >"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY SYSTEM
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT "DESCRIPTION:  ";D$
440 PRINT USING  "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
```

```
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)=0 THEN PRINT "ONLY ";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=CVI(R$) THEN PRINT "QUANTITY NOW ";Q%;
    " REORDER LEVEL ";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$) < CVI(R$) THEN PRINT D$;" QUANTITY ";
    CVI(Q$)TAB(50)"REORDER LEVEL ";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART% < 1) OR (PART% >100) THEN PRINT "BAD PART
    NUMBER":GOTO 840
860 GET#1,PART%
890 RETURN
900 REM INITIALIZE
910 INPUT "ARE YOU SURE ";B$:IF B$ <>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
970 CLOSE
```

5 BASIC's USE OF MEMORY

This chapter discusses the layout and use of the memory when BASIC is loaded. It includes a diagram of the memory divison and a breakdown of the relevant sections. There are various pointer addresses mentioned during the course of this chapter and you will find them listed in full in Appendix F. However, you should bear in mind that these addresses can change with each new release of BASIC.

There is also a manual available which covers all technical specifications of the P2000, including full details of the memory divisions which are not BASIC-dependent. Although some of the 'interesting' addresses mentioned in that manual are not relevant to BASIC itself, they can be accessed by BASIC. You should find these very useful.

It is worth noting that P2000 BASIC uses the inverse representation for values and addresses. This is the same for every Z80 configuration. For example a 2-byte integer is held as:

Address	Address +1
LOW	HIGH

So, a value of 3 would give -

00000011	00000000

The same rule applies to addresses, for instance a variable pointer positioned at Address and Address +1 which holds the following values -

Address	Address +1
00000000	11011111

is in fact pointing to Address &HDF00

5.1 MEMORY LAYOUT

```
         ______________________________  0000
           ROM
                    SYSTEM ROM
0FFF     ______________________________  1000
           ROM

           BASIC INTERPRETER PART I


4FFF     ______________________________  5000
           RAM

                 VIDEO MEMORY

5FFF     ______________________________  6000
          MONITOR/SYSTEM WORK AREA
61FF     ______________________________  6200

6380         INTERPRETER MEMORY +
           DATA STRUCTURE HANDLING

68FF     ______________________________  6900
             INTERFACE BETWEEN -
           BASIC AND MONITOR ROM
707F     ______________________________  7080
          BASIC INTERPRETER PART II
         ------------------------------      (TXTTAB)
                 SOURCE TEXT
         ------------------------------      (VARTAB)
                  VARIABLES
         ------------------------------      (STREND)

                   ? FRE(0)

         ------------------------------      (FRETOP)
                 STRING AREA
         ------------------------------      (MEMSIZ)
                    STACK
         ------------------------------
         FREE FOR USE OUTSIDE BASIC          If KSAM is in use, then
                                             it is loaded between :
                                             E500 - FFFF
FFFF     ______________________________
```

5.2 SOURCE TEXT

The source text of the BASIC program in memory can be located by the contents of the variable pointer TXTTAB (see Appendix F) and TXTTAB +1. This contains the start address of text.

Each line number within the program is made up in the following way :

```
------------------------------------------------------------
  A (2 bytes)  B (2 bytes)  STATEMENT............  0 (1 byte)
------------------------------------------------------------
```

A is a 2-byte pointer to the next line

B is a 2-byte binary line number

STATEMENT .... is the BASIC statement in encoded form

0 is a 1-byte end-of-line marker containing binary 0

The last line of the program is followed by 2 bytes which both contain binary zero. So in fact the end of program is indicated by 3 zero bytes.

So, the complete text layout can be represented like this:

```
 _______________________________________________________
| A  B  Statement...  0 A  B  Statement...  0 A  B ......



  ...............          0 0 0
 _______________________________________________________
```

5.3 VARIABLES AND ARRAYS

The part of memory used for Variables and Arrays can be seen like this:

VARTAB

Simple variables + values

String descriptors

ARYTAB

Arrays - names + values

5.3.1 Variables

A variable is made up in the following way:

Name + attributes   Value

VARPTR always returns this address

The value part of the variable can be one of four types:

- 2-byte integer
- 3-byte string descriptor
- 4-byte single precision floating-point number
- 8-byte double " " "

These are the formats of each of the value types:

2-byte integer

L	H

L is low value
H is high value

3-byte string d.

LGTH	LAD	HAD

LGTH is length of string
LAD is low address (of the string itself)
HAD is high address (")

4-byte single pr.

M3	M2	M1	E

M3 is mantissa 3
M2 is mantissa 2
M1 is mantissa 1
E is exponent

8-byte double pr.

M7	M6	M5	M4	M3	M2	M1	E

M7 - M1 are mantissa 7,6,5,4,3,2,1
E is exponent

Note - In all these types a VARPTR will point to the leftmost byte ie.

L	for the integer
LGTH	for the string descriptor
M3	for the single precision floating point number
M7	for the double precision floating point number

SINGLE AND DOUBLE PRECISION

Single precision and double precision numbers are represented in exactly the same way except that a single precision number has 3 bytes for the mantissa and a double precision has 7 bytes.

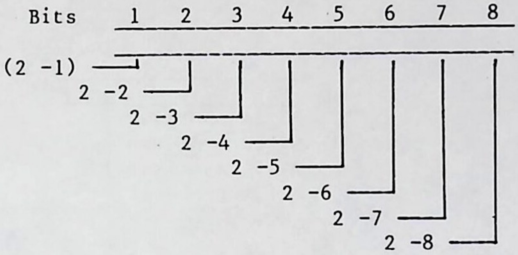
The internal representation is always:

M x 2 h where 0.5 M 1

The exponent is always equal to :

128 + h

Mantissa 1 represents 2 -1 through 2 -8 like this:



2 -1 is always implied (= 1 by normalisation) and so this bit is available as the sign bit.

Mantissa 2 represents 2 -9 through 2 -16
Mantissa 3 represents 2 -17 through 2 -24

and so on for mantissa 4 to mantissa 7

Examples:

- 00000000 00000000 00000000 10000010

  is (2 -1) x 2 2 = 2

- 00000000 00000000 10000000 10000010

  is (-2 -1) x 2 2 = -2

- 00000000 00000000 01010000 10000001

  is (2 -1 + 2 -2 + 2 -4) x 2 1 = 1.625

5.3.2 Arrays

An array variable is held like this:

Name+attributes	Array descriptor and size of Dim.	Value arr.elem (0,...,0)	arr.elem. ..(i, j)

VARPTR (A(m,...,n)) ________________________________

The value part is exactly the same as for a variable, of course depending upon the type of array that you have dimensioned.

Note - If a simple (not array) variable is declared explicitly or implicitly then all array variables shift; see memory layout. The VARPTR result will then change.

5.3.3 String Area

You should be aware that the string area is dynamically allocated by using the next available address within the string area. So, for example if you have used VARPTR to obtain the address of a string descriptor and hence the address of the string itself, the assignment of a new value will cause the address of the string to change. The value of the VARPTR does not change, only the address of the actual string value.

Also, when the string area has been filled, a re-organisation takes place which overwrites the old string values that are now redundant. The time taken to re-organise will of course depend on the size of the string area and how much of it is in current use. Again, most or even all of the addresses held in the string descriptors will change, so in fact it is advisable when using VARPTR to execute the function each time.

5.3.4 Sample Program

The following program list provides a simple but useful memory dump program. With it you can display on just the screen or on the screen and printer. All special characters which cannot be printed are replaced by a "." character.

The following page shows an example of output from this program.

```
10 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
20 DEFINT A-Z
30 INPUT "PRINTER (Y/N)    ",P$:P=P$="Y"
40 PRINT:INPUT "START ADDRESS OF DUMP (XXXX)    ",I$:Z$="&H":I$=Z$+I$:I=VAL(I$):P
RINT:PRINT
50 IF I=0 THEN I=J
60 PRINT "      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F"
70 IF P THEN LPRINT:LPRINT"      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F"
80 FOR J=I TO I+20*16-1 STEP 16
90 A=(J AND &HFF00)/256:GOSUB 240:A=J AND &HFF:GOSUB 210
100 FOR K=J TO J+15
110 A=PEEK(K):GOSUB 210
120 NEXT
130 IF P THEN LPRINT "   ";
140 PRINT "   ";:FOR K=J TO J+15:A=PEEK(K) AND 127:IF A<32 OR A=127 THEN A=46
150 IF P THEN LPRINT CHR$(A);
160 PRINT CHR$(A);:NEXT
170 IF P THEN LPRINT
180 PRINT
190 NEXT
200 I=J:GOTO 40
210 GOSUB 240:PRINT " ";
220 IF P THEN LPRINT " ";
230 RETURN
240 B=(A AND &HF0)/16:GOSUB 250:B=A AND 15
250 RESTORE:FOR C=0 TO B:READ A$:NEXT:PRINT A$;
260 IF P THEN LPRINT A$;
270 RETURN
```

5.3.5 Sample Output

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
6000 00 00 00 00 00 00 00 00 00 00 00 00 00 FF 31 00    ..............1.
6010 33 56 00 01 00 00 01 00 00 00 00 00 00 00 00 00    yV..............
6020 30 61 37 61 00 00 38 00 00 00 00 00 00 00 00 00    0a7a..8.........
6030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
6040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
6050 00 00 00 00 00 E2 FF 00 00 00 00 00 03 01 E0 6F    .....b........-o
6060 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
6070 00 A2 09 42 01 06 00 01 01 10 0E 00 03 0F 01 05    .".B............
6080 02 07 01 03 03 60 34 41 80 00 06 00 10 01 FE 61    .....-4A......÷a
6090 00 00 00 00 00 00 FF 0A 00 00 00 00 00 00 00 00    ................
60A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60B0 4F 30 57 0E 30 57 00 00 AE 94 00 10 8A 00 00 00    O0W;0W..........
60C0 00 00 00 00 00 00 20 FE 00 00 00 03 06 00 62 00    ...... ÷......b.
60D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
6100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
6110 00 00 05 01 09 01 B1 00 68 01 00 00 62 61 B1 00    ......1.h...ba1.
6120 B1 00 68 FF 00 00 62 61 7A E7 D1 E2 0A E0 44 10    1.h...bazg@b.-D.
6130 3E 01 D3 94 C3 A8 E9 F3 F5 3A 5F 61 3D 32 5F 61    >.S.C(isu:#a=2#a
```

5.4 MACHINE CODE SUBROUTINES

BASIC has provisions for interfacing with machine code subroutines. The USR Function allows machine code subroutines to be called in the same way BASIC´s intrinsic functions are called.

5.4.1 Memory Allocation

Memory space must be set aside for a machine code subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage.If more stack space is needed, BASIC´s stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC´s stack must be restored, however, before returning from subroutine.

5.4.2 USR Function Calls

The format of the USR funtion is

USR[digit] (argument)

where "digit" is from 0 to 9 and the argument is any numeric or string expression. "digit" specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If "digit" is omitted, USR0 is assumed. The adress given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two´s complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and FAC-2 contains the middle 8 bits of mantissa and FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative). FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Note - If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example:

```
A$ = "BASIC"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be an integer.

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L].

5.4.3 Interrupts

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied.

The assembly language subroutine may be loaded into memory by means of the BASIC POKE statement.

Example:

```
 10    LAST = &HF9FF
 20    CLEAR 300, LAST            REM SET HIGH MEMORY
 25    LAST = &HF9FF
 30    REM MOVE CURSOR
 40    DATA &H2A,&HB1,&H60:       REM LD HL,(60B1H)
 50    DATA &HED,&H4B,&HB3,&H60:  REM LD BC,(60B3H)
 60    DATA &H06,&H00:            REM LD B,0H
 70    DATA &H09      :           REM ADD HL,BC
 80    DATA &HCB,&HBE:            REM RES 7,(HL)
 90    DATA &HC9      :           REM RET
100    FOR I=LAST+32 TO LAST+44
110    READ N
120    POKE I,N
130    NEXT ;
140    DEF USR0 = LAST+32


170    REM SET LINE NUMBER 'LIN'  (RANGE 0 TO 24)
180    REM SET COLUMNNUMBER 'COL' (RANGE 0 TO 80)
190    REM CALL SUBROUTINE TO MOVE CURSOR
200    I = USR0 (I)
210    LIN = 20400 + 80 * LIN
220    HIB = INT (LIN/256)
230    LOB = LIN - 256 * HIB
240    POKE &H60B1,LOB: POKE &H60B4,LOB
250    POKE &H60B2,HIB: POKE &H60B5,HIB
260    POKE &H60B3,COL
270    REM
```

APPENDIX A RESERVED WORDS

ABS
AND
ASC
ATN
AUTO
CALL
CDBL
CHAIN
CHR$
CINT
CLEAR
CLOAD
CLOSE
COMMON
CONSOLE
CONT
COS
CSAVE
CSNG
CVD
CVI
CVS
DATA
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD
MKI
MKS
MOD
NAME
NEW
NEXT
NOT
NULL
OCT$
ON
OPEN
OPTION
OR
OUT
PEEK
POKE
POS
PRINT
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TO
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WEND
WHILE
WIDTH
WRITE
XOR

APPENDIX B SCREEN AND CURSOR HANDLING

BASIC has provisions for the screen's I/O management. These procedures are examined in this appendix.

I1. Clear Screen, Cursor Home

With PRINT CHR$(12), the screen is cleared and the cursor goes to the home position.

I2. Ring the Bell

With PRINT CHR$(7), the bell rings.

I3. Move Cursor

To move the cursor to line and position, execute PRINT CHR$(4); CHR$(line); CHR$(position). The allowable range for "line" is 1 through 24 and for "position", 1 through 80.

I4. Move Cursor to the Beginning of the Next Line

PRINT

I5. Print Screen

The screen's image is routed to the printer by PRINT CHR$(5).

I6. Change Screen Output Speed

At power-up, or after a system reset, the screen is set to the highest output speed which is indicated by a 0 (zero byte) on location &H6090.

You can slow the output speed by resetting it with, for example, POKE &H6090,8.

The output speed of the screen slows down proportionally to this value until another POKE to the address is executed. The lowest possible output speed is set by POKE &H6090,255.

APPENDIX C SUMMARY OF ERROR CODES AND ERROR MESSAGES

Number	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.)
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of DATA A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result off: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent 5. a call to a USR function for which the starting address has not yet been given 6. an inproper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
6	Overflow The result of a calculation is too large to be represented in BASIC number format. If underflow occurs, the result is zero and execution continues without an error.
7	Out of memory A program is too large, has too many FOR loops or GOSUB's, too many variables, or expressions that are too complicated.

APPENDIX C SUMMARY OF ERROR CODES AND ERROR MESSAGES

Number	Message
8	Undefined line number A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.
9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
10	Duplicate Definition Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
11	Division by zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
14	Out of string space String variables exeed the amount of allocated string space. BASIC will allocate 50 bytes of stringsspace, unless string space is allocated by the CLEAR statement.
15	String too long An attempt is made to create a string more than 255 characters long.
16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.

APPENDIX C SUMMARY OF ERROR CODES AND ERROR MESSAGES

Number	Message
17	Can't continue An attempt is made to continue a program that: 1. has halted due to an error, 2. has been modified during a break in execution, or 3. does not exist.
18	Undefined user function A USR function is called before the function definition (DEF statement) is given.
19	No RESUME An error trapping routine is entered but contains no RESUME statement.
20	RESUME without error A RESUME statement is encountered before an error trapping routine is entered.
21	Unprintable error An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt is made to input a line that has too many characters.
26	FOR without NEXT A FOR was encountered without a matching NEXT.
29	WHILE without WEND A WHILE statement does not have a matching WEND.
30	WEND without WHILE A WEND was encountered without a matching WHILE
31	Printer error
50	FIELD overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal error An internal malfunction has occurred in Philips BASIC. Report to Philips.

APPENDIX C SUMMARY OF ERROR CODES AND ERROR MESSAGES

Number	Message
52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	File not found A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.
54	Bad file mode An attempt is made to use PUT, GET or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O or R.
55	File already open A sequential output mode OPEN is issued for a file that is already open - or a KILL is given for a file that is open.
57	Disk I/O error An I/O error occurred which is fatal; i.e. the operating system cannot recover from the error.
58	File already exists The filename in a NAME statement already exists.
61	Disk full All disk storage space is in use.
62	Input past end An INPUT statement after all the data in a file has been input, or for an empty file. To avoid this error, use the EOF function to detect the end of file.
63	Bad record number In a PUT or GET statement, the record number is greater than the maximum (32767), or equal to zero.
64	Bad file name An illegal form of filename used with LOAD, SAVE, KILL or OPEN (e.g. a name with too many characters).
66	Direct statement in file A direct statement is met while loading an ASCII-format file. The LOAD is terminated.
67	Too many files An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX D DERIVED FUNCTIONS

Functions that are not intrinsic to BASIC may be calculated as follows:

Function	BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +((SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=-ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=-EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character	ASCII Code	Charac.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	'
006	ACK	049	1	092	'
007	BEL	050	2	093	'
008	BS	051	3	094	↑
009	HT	052	4	095	#
010	LF	053	5	096	–
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	£
038	&	081	Q	124	‖
039	'	082	R	125	¾
040	(	083	S	126	÷
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Note - Some of these codes are only applicable to the P2000.

APPENDIX F USEFUL ADDRESSES

There are a number of pointer variables which are used by BASIC. Each one is the start address of a 2-byte variable and each variable contains an address within the User area. Some of these pointers are mentioned in Chapter 5, however you should be aware that their locations can change with new releases of BASIC.

Remember the Inverse representation when reading the contents of the pointers.

POINTER	ADDRESS	POINTS TO:
TXTTAB	625C	Beginning of BASIC source text
VARTAB	6405	Start of simple variable space
ARYTAB	6407	Beginning of Array space
STREND	6409	End of storage in use
FRETOP	63DD	Top of free string space
MEMSIZ	63B8	Highest location in memory
DATPTR	640B	Start of Data area
BUF	6261	Keyboard input buffer (length 256)

APPENDIX G KEYWORD SUMMARY

This appendix describes each of the keywords briefly. The headings are shown as follows:

Keyword Type Description

The type is denoted by one of the following:

AF	Arithmetic Function
SF	String Function
MF	Miscellaneous Function
O	Operator
C	Command
S	Statement

Each keyword has a short one line description and a second line showing the format.

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
ABS	AF	Absolute value of expression
AND	O	Bitwise AND (integer)
ASC	SF	Return ASCII value of the first character of a string
ATN	AF	Arctangent of the expression (result in radians)
AUTO	C	Generate program line numbers automatically
CALL	S	Call a Z80 machine-code subroutine
CDBL	AF	Convert the expression to a double precision number
CHAIN	S	Call (or merge) new program and pass variables to it
CHR	SF	Returns a character which has the ASCII code of exp
CINT	AF	Convert the expression to an integer
CLEAR	C	Clear prog'm variables, set end of memory & stack space

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
CLOSE	S	Close disk files (or all open ones if no argument)
COMMON	S	Define variables to be passed to a CHAINed program
CONT	C	Continue program execution
COS	AF	Return the cosine of exp (exp is specified in radians)
CSNG	AF	Convert the expression to a single precision number
CVD	SF	Convert an 8-byte string to a double precision number
CVI	SF	Convert a 2-byte string to an integer
CVS	SF	Convert a 4-byte string to a single precision number
DATA	S	List data to be used in a READ statement
DEF	S	Define function or address of machine code routine
DEFDBL	S	Define all letters to be double precision variables

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
DEFINT	S	Define all letters to be integer variables
DEFSNG	S	Define all letters to be single precision variables
DEFSTR	S	Define all letters to be string variables
DELETE	C	Delete program lines
DIM	S	Allocate space for arrays, set maximum subscript values
EDIT	C	Edit a program line
ELSE		Keyword used with the IF statement
END	S	Stop program, close files and return to command level
EOF	AF	Returns true or false for End Of File check
EQV	O	Bitwise equivalence

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
ERASE	S	Release space & variable names reserved for arrays
ERL		Keywords related to error trapping routines
ERR		Error line in ERL - Error code in ERR
ERROR	S	Generate an error - 'code'
EXP	AF	Raise the constant e to the power of expression
FIELD	S	Define fields in a random buffer area
FILES	C	List files in disk directory
FIX	AF	Returns truncated integer of expression
FN		Keyword used with DEF to define a user function
FOR	S	Used with NEXT to repeat a sequence of program lines
FRE	SF	Return remaining free space (maybe with string re-org)

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
GET	S	Read a record from a random disk file (See FIELD,PUT)
GOSUB	S	Call a BASIC subroutine by branching to line number
GOTO	S	Branch to a specified line number
HEX$	SF	Convert a number to a hexadecimal string
IF	S	Conditional check used with 'THEN... ELSE' and 'GOTO'
IMP	O	Bitwise implication (integer)
INKEY$	MF	Returns a single character string from keyboard input
INPUT	S	Read data from keyboard with display on monitor
INSTR	SF	Returns first position of Y$ in X$ from offset pos'n I

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
INT	AF	Evaluates exp for the largest integer contained
KILL	C	Delete a disk file
LEFT$	SF	Returns a string of the leftmost I characters of X$
LEN	SF	Returns the length of string X$
LET	S	Assign a value to a variable
LIST	C	List program lines on monitor
LLIST	C	List program lines on printer
LOAD	C	Load a program file (,R option means RUN)
LOC	MF	Returns the current record number of file number

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
LOG	AF	Returns the natural logarithm of x
LPOS	C	Returns the current position of the line printer print head
LPRINT	C	Lists data on printer
MID	SF	Replaces part of one string with another
MOD	O	Arithmetic operator
NEW	C	Deletes the current program
NEXT	S	Used with FOR to repeat a sequence of program lines
NOT	O	Relational operator
OCT	SF	Returns a string representing the octal value of the decimal argument
ON	C	Specifies the start of an error-handling routine

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
OR	O	Relational operator
OUT	C	Sends a byte to a machine output port
PEEK	AF	Returns the value of a memory byte
POKE	C	Writes a byte into memory
POS	MF	Returns the current cursor position
PRINT	C	Prints data at the terminal
READ	C	Reads values from a DATA statement
REM	S	Inserts a remark into a program
RESTORE	C	Allows DATA statements to be re-read from a specified point
RESUME	C	Returns from error routine
RETURN	C	Returns from a subroutine
RIGHT	SF	Returns last (n) characters of a string

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
RND	AF	Returns a random number between 0 and 1
RUN	C	Executes current program in memory
SGN	AF	Returns -1 for negative expressions, 0 for zero and +1 for positive
SIN	AF	Gives the sine of expression in radians
SPACES	SF	Returns a string of spaces
SPC	MF	Prints blanks on the terminal
SQR	AF	Returns the square root of an expression
STEP	S	Used with FOR...NEXT statements to specify number of executions
STOP	C	Stops program execution
STR	SF	Returns a string representation of a specified value
STRING	SF	Returns a string of symbols having the same first character as x$
SWAP	C	Exchanges the values of two variables

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
TAB	C	Spaces to position on current output line
TAN	AF	Returns the tangent of an expression in radians
THEN	S	Conditional check used with IF statement
TO	S	Used with FOR to repeat a sequence of program lines
TROFF	C	Stops execution of program trace
TRON	C	Starts execution of program trace
USING	S	Used with the PRINT statement to print strings
USR	MF	Branches to machine language subroutine
VAL	AF	Returns the numeric value of a string
VARPTR	SF	Returns the address of the first byte of specified data

APPENDIX G KEYWORD SUMMARY

Keyword	Type	Description
WAIT	C	Stops program to check port status
XOR	O	Logical operator

APPENDIX H PROGRAM EFFICIENCY

The efficiency of P2000 Disk BASIC can be improved by following a few simple guidelines. The speed at which a program is processed depends on the complexity of the operations and the number of instructions. In most simple programs, speed is not important. But when you write longer and more intricate programs, speed may become an important factor. Here are some suggestions to help you design more efficient programs.

1. Remove all unnecessary lines in the program (REM statements, etc.)

2. Do not use unnecessary spaces between statements, operators, etc.

3. Use multiple-statement program lines whenever possible (separating each two statements with a colon). Every combined statement saves 4 bytes.

4. It is important to use constants rather than variables in operations. The P2000 encodes constants directly in internal source code, whereas variables have to be searched for in a table.

5. Use integer variables whenever possible. Integer arithmetic is faster than floating point arithmetic. Also, integer values take up less space than floating point values (2 bytes instead of 4 for single precision and 8 bytes for double precision).

6. You will save program space by using subroutines if certain program sequences appear in more than one place in your program.

7. Arrays should be DIMensioned carefully.

APPENDIX H PROGRAM EFFICIENCY

8. Use DEF statements when you work with values that are not single precision; i.e. strings, integers and double precision values. Then you will save space because you will not need type declaration characters with the variable name references.

9. Define the most commonly-used variables first (for example, by assigning a dummy value to them). When you define a variable, it is positioned at the top of the variable table. The second variable will be below the first. When you access a variable, the table will be searched sequentially from the top. In this way, you will save time.

10. Put DATA statements at the end of the program.

11. Put those subroutines that are most commonly used at the beginning of the program.

12. The conditional jump which is implied at the end of a FOR... NEXT loop, knows where the statement it is going to resides in memory. So, if you have loop-like constructions programmed with an explicit conditional jump backwards, you can prevent the time-consuming action of searching for a line by inserting, at the beginning and end of such a loop, a FOR...NEXT construction.

13. For performance reasons, it is recommended that all COMMON statements be placed at the end of the program.

APPENDIX I THE VOLORG UTILITY

The Volume Organization Utility supports the field-development of BASIC user programs. It resides on Disk P2306 or P2311 (PART 3). The utility is invoked by the following command:

```
RUN "B:VOLORG"
```

The following menu is shown:

```
P 2000    D I S K   U T I L I T Y


0  =  LOAD AND RUN PROGRAM
1  =  READ DISK DIRECTORY AND FREE SPACE
2  =  COPY FILE
3  =  COPY DISK
4  =  DELETE FILE
5  =  DELETE DISK
6  =  RENAME FILE
7  =  SET / RESET FILE PROTECT
8  =  FILE SIZE
9  =  COMPARE DISK
A  =  INPUTS FOR AUTO-LOAD
E  =  EXIT TO COMMAND MODE
R  =  RESET AFTER DISK CHANGE OR ERROR


SELECT
```

Note - Writing on a disk is only possible if the write-protect label is removed and if the disk does not contain a write-protected program file.

A short description of these functions follows.

APPENDIX I THE VOLORG UTILITY

0 LOAD AND RUN PROGRAM

This function has the same effect as typing "E" (EXIT TO COMMAND MODE), followed by:

RUN "program name".

1 READ DISK DIRECTORY AND FREE SPACE

This function displays the first 64 directory entries and the free space in kilobytes (= 1024 bytes). Typing a "P" causes the screen to be printed. Typing a "1" shows the other 64 directories alternately.

An asterisk behind the name shows that the file has been protected by VOLORG.

Note - For each 16k byte part of a file, a directory entry is created.

2 COPY FILE

To copy a file from one disk to the same disk (a different file name must be used) or from one disk to another (the same or a different name may be used).

3 COPY DISK

To copy a disk from Drive A to Drive B.

Note - The function can only be executed if the disk to be copied is not a system disk and if it does not contain a file protected by VOLORG.

4 DELETE FILE

To delete a file.

APPENDIX I THE VOLORG UTILITY

5 DELETE DISK

To delete the contents of a disk.

6 RENAME FILE

To rename a file (filename + filename extension).

7 SET / RESET FILE WRITE PROTECT

- To write-protect a program file. This function causes the whole disk to be write-protected.
- To write-protect a data file.
- To write-enable a data file.

8 FILE SIZE

To show the length of a file in sectors (1 sector = 256 bytes).

9 COMPARE DISK

This function compares the disks in Drive A and B and reports whether they are equal or not.

APPENDIX I THE VOLORG UTILITY

A INPUTS FOR AUTO-LOAD

With this option, the following parameters can be given to the system:

- The number of files to be processed simultaneously. It implies, also, the maximum file number used in BASIC programs.
- Runtime support. A "Y" means that during System Reset/Power On, the following subroutines are loaded:
  1. From address F800 upwards - FIELD ENTRY (P2306 and P2311).
  2. From address E500 to F7FF - KSAM (P2311 only).
- The program to be loaded and run automatically after System Reset/Power Up.

E EXIT TO COMMAND MODE

To leave the VOLORG program.

R RESET AFTER DISK CHANGE OR ERROR

After a disk change, it is necessary to tell the system to read the administration (directory) of the newly-inserted disk. After an error, VOLORG itself gives the RESET command (see also the RESET command in Chapter 3).

INDEX

page

ABS 3.3
Address - Useful Start Addresses F.1
Array 5.7
Array Variable 2.8
ASC 3.3
ASCII - Character Codes E.1
ATN 3.4
AUTO 3.5
    Auto-line Feed 2.2

CDBL 3.7
Character Set 2.3
CHR$ 3.10
CINT 3.10
CLEAR 3.11
CLOSE 3.12
    Security Copy 1.1
    Interpreter 1.3
Command 3.1
Compiler
    Theory 1.2
    Relationship 1.3
    Diagram 1.3
Constants
    String, Numeric 2.5
    Integer 2.5
    Fixed Point 2.5
    Floating Point 2.6
    Hexadecimal 2.6
    Octal 2.6
    Single Precision 2.6, 2.7
    Double Precision 2.6, 2.7
COMMON 3.13
CONT 3.14
COS 3.15
CSNG 3.16
Cursor Handling 3.1

page

CVD 3.17
CVI 3.17
CVS 3.17

DATA 3.18
Declaration Character 2.7
DEF DBL 3.21
DEF FN 3.19, 3.20
DEF INT 3.21
DEF SNG 3.21
DEF STR 3.21
DEF USR 3.22
DELETE 3.23
Dimension 2.8
DIM 3.24
Division by Zero 2.12

EDIT 3.25
    Mode Subcommands 3.25, 3.26, 3.27
    Input Editing 2.16
    Syntax Errors 3.28
Editor 1.3
END 3.29
EOF 3.30
ERASE 3.31
ERL 3.32
ERR 3.32
ERROR 3.33, 3.34
Error - Codes and Messages C.1, C.2, C.3, C.4
EXP 3.35

FIELD 3.36
FIX 3.37
FN
    DEF FN 3.19
    With Variable 2.7
Format Notation 3.1
FOR...NEXT 3.38, 3.39
FRE 3.40
Function 3.1, 3.2
Function Derived D.1

GET 3.41
GOSUB 3.42
GOTO 3.43

HEX$ 3.44

IF 3.45, 3.46
INKEY$ 3.47
INP 3.48
INPUT 3.49, 3.50
INPUT# 3.51
INPUT$ 3.52
INSTR 3.53

page

INT 3.53
Interpreter
  Execution of BASIC 1.2
  Diagram 1.3

KILL 3.54, 4.4
Keyword - Summary G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8, G.9, G.10, G.11, G.12

LEFT$ 3.55
LEN 3.55
LET 3.56
Line Format 2.1
Line Format 2.2
LINE INPUT 3.57
LINE INPUT# 3.58
Line Number 2.2
LIST 3.59, 3.60
LLIST 3.61
LOAD 3.62, 4.3
LOC 3.63
LOG 3.63
Loop - Nested 3.38, 3.39
LPOS 3.65
LPRINT - USING 3.66
LSET and RSET 3.67

Memory 5.1
MERGE 3.68, 4.4
MID$ 3.69, 3.70
Mode
  Direct 2.1
  Indirect 2.1

NAME 3.72, 4.4
NEW 3.73

Object Module 1.2
OCT$ 3.74
ON ERROR 3.75
ON GOSUB 3.76
ON GOTO 3.76
Operator
  Arithmetic 2.10, 2.11, 2.12
  Functional 2.15
  Logical 2.13, 2.14
  Relational 2.12
OPEN 3.77
OPTION BASE 3.78
OUT 3.79
Overflow 2.12

P. Code Interpreter 1.3
PEEK 3.79

page

POKE 3.80
POS 3.81
PRINT 3.82, 3.83
PRINT USING 3.84, 3.85, 3.86, 3.87
PRINT# AND PRINT# USING 3.88, 3.89
Program Efficiency H.1, H.2
PUT 3.90

RANDOMIZE 3.91
READ 3.92, 3.93
REM 3.94
Reserved Words A.1
RENUM 3.95
RESET 3.96
RESTORE 3.97
RESUME 3.98
RETURN 3.99
RIGHT$ 3.100
RND 3.100
RUN 3.101, 4.4

SAVE 3.103, 4.4
SGN 3.104
SIN 3.104
Source Module 1.2
Source Text 4.2
SPACE$ 3.105
SPC 3.105
SQR 3.106
STOP 3.107
String Area 5.6
    Operation 2.15
    Variable Name 2.8
STRING$ 3.108
STR$ 3.108
Subscript 2.8
Subroutine Machine Code 5.9, 5.10, 5.11
SWAP 3.109

TAB 3.110
TAN 3.110
TROFF 3.111
TRON 3.111
Type Conversion 2.9, 2.10

USR 3.112
USR - Function 4.10, 4.11

VAL 3.112
Variable 2.7, 2.8, 2.9, 5.4, 5.5, 5.6
VARPTR 3.113
VOLORG C.1, C.2, C.3, C.4

	page
WAIT	3.113
WEND...WHILE	3.114
WIDTH	3.116
WRITE	3.117
WRITE	3.118

DISK BASIC REFERENCE MANUAL

12NC: 5103 991 10625

This publication comprises the following updates:

no updates

DISK BASIC REFERENCE MANUAL

12NC: 5103 991 10625

Including update(s).........

Originator:

Name ....

Address ....
....
....
....

Comment (if possible, add a copy of the page(s) affected by the comment, marked with the proposed changes).

Please return this form to: Osterr. Philips Ind. Ges.m.b.H.
MAG-PERCO / MARKETING SUPPORT
A-1100 Wien, Triesterstr. 64,
AUSTRIA